Algorithms for Constructing Exact Nearest Neighbor Graphs

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

David C. Anastasiu

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Dr. George Karypis, Advisor

June, 2016

Acknowledgements

A great deal of time and effort was invested in producing this thesis, and none of it would have been possible without the help and support of family, friends, mentors, and teachers.

First and foremost, I would first like to thank my advisor, George, for believing in me even when I did not believe in myself, and for teaching me the importance of hard work and perseverance. He has spent a great deal of time helping me grow as a researcher and as an engineer. I am deeply indebted to him, and consider myself very lucky to have had such a talented advisor.

I would like to thank Professors Sen Arnab, Arindam Banerjee, John Carlis, and Georgios Giannakis for taking the time to serve on my preliminary exam and thesis committees. Furthermore, I am thankful to my Masters advisor, Byron J. Gao, without whose initial guidance and encouragement I would not have even considered pursuing a PhD degree.

I owe a great deal of thanks to the past and present members of Karypis Lab: Asmaa, Agi, Chris, Dominique, Eva, Fan, Haoji, Jeremy, Kevin, Mohit, Rezwan, Santosh, Sara, Saurav, Shaden, and Xia. They have helped me in more ways than one to cope with the ups and downs of graduate school and research activities. They provided a constant source of encouragement, support, and laughter. I am grateful to have known each and every one of them, and I will miss them all dearly.

I would like to thank the wonderful staff at the Department of Computer Science, the Digital Technology Center, and the Minnesota Supercomputing Institute at the University of Minnesota for providing assistance, facilities, and other resources for my research. I would also like to thank the Graduate School at University fo Minnesota for generously funding my research activities during my last year there.

I would like to thank my family, Mariana, Monica, and Miron, for taking a leap of faith and letting me pursue my dreams half way across the globe. Without their steadfast support and selfless commitment to my future and my happiness I would not be where I am today.

Last, but definitely not least, I would like to thank my wife, Heather, and my son Joseph. They are my light at the end of the tunnel, my happy at the end of the day, my *joie du vivre*.

Dedication

To my wife, my constant encourager and love of my life.

Abstract

Nearest neighbor graphs (NNGs) contain the set of closest neighbors, and their similarities, for each of the objects in a set of objects. They are widely used in many real-world applications, such as clustering, online advertising, recommender systems, data cleaning, and query refinement. A brute-force method for constructing the graph requires $O(n^2)$ similarity comparisons for a set of n objects. One way to reduce the number of comparisons is to ignore object pairs with low similarity, which are unimportant in many domains. Current methods for construction of the graph tackle the problem by either pruning the similarity search space, avoiding comparisons of objects that can be determined to not meet the similarity bounding conditions, or they solve the problem approximately, which can miss some of the neighbors.

This thesis addresses the problem of efficiently constructing the exact nearest neighbor graph for a large set of objects, i.e., the graph that would be found by comparing each object against all other objects in the set. In this context, we address two specific problems. The ϵ-nearest neighbor graph (ϵ-NNG) construction problem, also known as all-pairs similarity search (APSS), seeks to find, for each object, all other objects with a similarity of at least some threshold ϵ. On the other hand, the k-nearest neighbor graph (k-NNG) construction problem seeks to find the k closest other objects to each object in the set. For both problems, we propose filtering techniques that are more effective than previous ones, and efficient serial and parallel algorithms to construct the graph. Our methods are ideally suited for sparse high dimensional data.

We address the ϵ-NNG construction problem for two similarity functions widely used in the data mining and chemoinformatics communities, cosine and Tanimoto. Our solution uses a number of novel bounds on the similarity of two vectors, based on their length, to filter those object pairs that will not be similar enough. We prove the effectiveness of the new filtering bounds, both theoretically and experimentally, and compare the efficiency of our methods against several state-of-the-art baselines for a range of ϵ values. Our methods achieve 2–13x lower runtimes than the best state-of-the-art alternative, in many cases outperforming even approximate methods required to obtain at least 95% of the correct result.

Next, we design a new algorithm that applies filtering techniques in a novel way to construct the k-NNG for a set of objects. Our method quickly builds an approximate solution to the problem, identifying many of the most similar neighbors, and then uses theoretic bounds on the similarity of two vectors, based on the ℓ^2-norm of part of the vectors, to find each object's exact k-neighborhood. We perform an extensive evaluation of our algorithms, comparing against both exact and approximate state-of-the-art baselines, and demonstrate the efficiency of our method across a variety of real-world datasets and neighborhood sizes. Our approximate method achieves high recall in less time than competing approximate state-of-the-art baselines, and is an order of magnitude more efficient when building a graph that is at least 95% correct. Furthermore, our exact method achieves 2–28x lower runtimes than exact state-of-the-art baselines.

Finally, we develop filtering based shared memory parallel methods for both the ϵ-NNG and the k-NNG construction problems. The pruning process in filtering based methods results in unpredictable memory access patterns that can reduce search efficiency. Our parallel graph construction methods use a number of cache-tiling optimizations, combined with fine-grained dynamically balanced parallel tasks, to solve the problem up to two orders of magnitude faster than existing parallel baselines, on datasets with hundreds of millions of non-zeros. In particular, our parallel ϵ-NNG method outperforms baselines using 24 cores by 2–232x. Using 16 cores, our parallel k-NNG method constructs an approximate graph containing at least 95% of the correct result 2–22x faster than previous methods, and is able to find all exact nearest neighbors in 3–13x less time than the best alternative.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Efficiently identifying nearest neighbors for large sets of objects has long been a challenging problem. Many data mining tasks must first identify nearest neighbors before their analysis can be completed. A number of data structures and efficient search algorithms exist for objects embedded in a small dimensional space. Finding neighbors for objects described by sparse vectors in a high dimensional space is a much harder open problem, which is often solved by approximation, returning only some of the nearest neighbors. In this thesis, we design novel data structures and algorithms that efficiently solve the exact nearest neighbor graph construction problem, which is akin to finding the *nearest* neighbors amongst a set of objects for each object in the set. We show that our solutions can be applied with multiple similarity functions, on data from several domains, and result in substantial gains in terms of runtime in comparison to previous state-of-the-art approaches. Furthermore, we provide efficient multi-core algorithms for solving the problem which scale well with an increasing number of cores and outperform previous baselines by an order of magnitude.

1.1 Problems & Applications

Computing the nearest neighbor graph, or similarity graph, for a set of objects is a common task in fields such as clustering [1, 2], online advertising [3], recommender systems [4], data cleaning [5, 6], query refinement [7, 8], and drug discovery [9]. Many algorithms in these fields use the type of methods described in this thesis to identify

nearest neighbors before performing their task. For example, item-based nearest neighbor collaborative filtering algorithms recommend items (e.g., books or movies) to a user based on the k most similar items to each of the user's preferred items [10]. In the context of data cleaning, near-duplicate objects can be detected by constructing a threshold-based nearest neighbor graph and merging or removing all but one object in each neighborhood. In the chemoinformatics domain, fueled by the generally valid premise that structurally similar molecules exhibit similar binding behavior and have similar properties [11], many methods use the computation of pairwise similarities as a kernel within their algorithms. Virtual screening (VS), for example, uses nearest neighbor based search, clustering, classification, and outlier detection to identify structurally diverse compounds that display similar bioactivity, which form the starting point for subsequent chemical screening [12, 13].

Objects in the real-world are often depicted as points in a high-dimensional feature space, numerically represented by vectors, where each dimension quantifies a relevant object feature. When only the presence of features is of interest, binary vectors suffice to encode the set of features in an object, each vector element indicating the presence (1) or absence (0) of a feature. However, weighted vectors often better represent objects for comparison [14, 15] and are standard in fields like information retrieval [16] and text mining [17]. In many domains, only some of all possible features are relevant for a given object, resulting in vectors with more zero than non-zero values, also called *sparse* vectors.

While a true measure of similarity between objects may not be feasible to obtain, a number of similarity functions have been devised that can be used to compare objects represented as points in $\mathbb{R}^m$, where m is the number of features or attributes. The choice of similarity function is often domain and data dependent. In general, it is assumed that a practitioner (e.g., data analyst) is familiar with what functions work well for the type of data they are analyzing. In this thesis, we address two popular similarity functions that have been shown to be very effective at different analysis tasks on various types of data: cosine and Tanimoto. Cosine similarity measures the cosine of the angle between the high dimensional vectors representing the objects and is a standard way to measure proximity of documents in text analysis or user/item profiles in collaborative filtering methods. Tanimoto similarity, on the other hand, is most often used in domains, such as

plagiarism detection or chemical compound search, where the amount of feature overlap in relation to the overall presence of features in the objects is important.

A naïve approach to constructing the nearest neighbor graph executes $O(n^2)$ object comparisons for a set of n objects. One way to reduce the number of comparisons is to ignore unimportant object pairs, i.e., those with low similarity. The problem is then to efficiently find, for each object in the set, those other objects with the highest similarity (smallest distance) values. Given a set of n objects $D = \{d_1, d_2, \ldots, d_n\}$, a nearest neighbor graph $G = (V, E)$ consists of a vertex set V, corresponding to the objects in D, and an edge set E which is a subset of VV. An edge (v_i, v_j) indicates that the jth object is similar *enough* to the ith object, i.e., that the jth object is in the ith object neighborhood. Based on how the similarity between two objects is bounded, there are two nearest neighbor graphs often used in practice:

- The ϵ-nearest neighbor graph (ϵ-NNG) contains an edge for each pair (v_i, v_j) with a similarity value above a predefined threshold $\epsilon \in \mathbb{R}^+$. When the similarity function being used is commutative, which is often the case, this results in an undirected graph. The problem of constructing the ϵ-NNG is also known as the *all-pairs similarity search* (APSS) or the *similarity join* problems.

- The k-nearest neighbor graph (k-NNG) contains an edge for each pair (v_i, v_j) when the similarity value $\text{sim}(d_i, d_j)$ between the ith and jth objects is among the k highest values in the set $\{\text{sim}(d_i, d_l) \mid l = 1, \ldots, i-1, i+1, \ldots, n\}$. The k-NNG is generally a directed graph.

1.2 Emerging Challenges

While the nearest neighbor graph construction problem is not new, and many solutions exist for the problem, the scale that the problems is applied to has dramatically increased over the years. Collaborative filtering systems from major companies such as Walmart, Amazon, or Netflix store profiles for tens of millions of users and hundreds of millions of items. Analyzing web traffic to identify potential fraudulent user rings involves sifting through tens of millions of user click sessions. The number of commercially available chemical compounds (currently $\sim 210^7$) is steadily increasing. We have now entered the

era of Big Data, which promises to deliver game changing effects in retail, science, and healthcare, among others, through the collection and analysis of huge volumes of data.

Given the ever increasing problem size, analysts must be able to effectively sift through the myriad of possibilities to find the few nearest neighbors needed to gain insight from such data. Existing methods do not scale to more than a million objects, taking hours or even days to provide an answer for problems at that scale. Methods are needed that can quickly and safely ignore objects that theoretically cannot be one of the nearest neighbors, providing a fast solution whose quality does not need to be questioned.

The trend in today's computer systems is no longer to increase processing speed, but rather the number of processing cores. While the number of cores is quickly climbing, the amount of memory that can be shared among the cores is slower to grow. Nearest neighbor graph construction methods must be able to efficiently execute on all available cores in a modern processor while making efficient use of often reduced memory available for each core. Given the large existing gap in speed between memory transfers and processor computations, this can only be achieved by designing methods that promote data reuse once it has been copied to the processor cache.

Solving nearest neighbor graph construction problems involving hundreds of millions of objects generally involves breaking the problem into a number of smaller problems and using many nodes in a supercomputer or the cloud to solve the sub-problems concurrently. The most efficient solutions for these problems usually involve efficient multi-core graph construction methods such as the ones described in this thesis. Advancements in designing multi-core graph construction methods will thus provide immediate benefits to distributed algorithms for solving the problem.

1.3 Contributions

The contributions of this thesis are the development of effective and efficient serial and shared memory parallel algorithms for constructing nearest neighbor graphs. We propose new theoretic bounds on the similarity of two vectors and algorithms that effectively use these bounds to efficiently deliver the exact solution to the problem. We show that both our serial and parallel algorithms achieve substantially better performance

than previous state-of-the-art methods.

1.3.1 ϵ-Nearest Neighbor Graph Construction

In the context of filtering based exact methods for ϵ-NNG construction, we provide a unifying framework which helps connect and explain previous state-of-the-art methods for solving the problem [18]. We then introduce new filtering strategies that allow the exact ϵ-NNG construction problem to be solved efficiently for cosine similarity and non-negative real-valued vectors. Our method uses upper bound estimates on the similarity of two vectors, after comparing only a few of their features, to filter those object pairs that will not be similar enough. We prove theoretically that the bounds we propose are tighter than previously proposed bounds, and show experimentally that our method effectively uses these bounds. We analyze the filtering process and find that our bounds lead to fewer object comparisons and non-neighbor objects being eliminated from consideration quicker than in previous approaches. As a result, our method is able to solve the problem 2–13x faster than the best alternative, depending on the input threshold ϵ, and up to 1600x times faster than a linear search. While baseline algorithms do not scale well as the similarity threshold decreases, our new pruning techniques make our method effective at both high and low similarity thresholds. In many of the experiments, our exact graph construction method is able to outperform even approximate methods required to obtain at least 95% of the correct result.

In the context of Tanimoto similarity, we show how cosine bounds we defined in [18] can be combined with new bounds and filtering techniques based on the length of vectors to solve the problem efficiently [19]. We define a new class of length-based bounds and show that a previously proposed bound [20] is actually the upper limit of the bounds we describe. We prove the effectiveness of our filtering bounds and compare the efficiency of our method against several state-of-the-art baselines for a range of ϵ values. Our method is up to 12.5x more efficient than the most efficient baseline and up to two orders of magnitude faster than a linear search. In particular, it was able to find all near-duplicate pairs among 5M chemical compounds in minutes, using a single CPU core.

1.3.2 k-Nearest Neighbor Graph Construction

We introduce a novel method for constructing the cosine k-NNG [21]. Our method uses an initial approximate solution graph as a guide to find the nearest k neighbors, through a modified similarity search framework. In this framework, we introduce several new pruning bounds specific to the k-NNG construction problem, which leverage the Cauchy-Schwarz inequality in partial vector dot-products at each stage in the framework to prevent full similarity computation for most object pairs. We perform an extensive evaluation of our algorithm, comparing against both exact and approximate baselines, and demonstrate the efficiency of our method across a variety of real-world datasets and neighborhood sizes. Our inexact k-NNG construction method achieves high recall in less time than competing approximate methods, and is an order of magnitude faster than our approximate baselines. Furthermore, our exact method computes fewer object similarities in full and is able to achieve an order of magnitude improvement against exact baselines.

1.3.3 Parallel Graph Construction Methods

We develop filtering based shared memory parallel methods for both the ϵ-NNG [22] and the k-NNG [23] construction problems. The pruning process in filtering based methods results in unpredictable memory access patterns that can reduce search efficiency. Our parallel graph construction methods use a number of cache-tiling optimizations, combined with fine-grained dynamically balanced parallel tasks, to solve the problem up to two orders of magnitude faster than existing parallel baselines, on datasets with hundreds of millions of non-zeros. In particular, our parallel ϵ-NNG method displays less than 2% load imbalance amongst the threads and has better scaling characteristics than all baselines. Our algorithm finds the exact solution, using 24 cores, 1.5–232x faster than the best alternative. Using 16 cores, our parallel k-NNG method constructs an approximate graph containing at least 95% of the correct result 1.5–21.7x faster that previous methods, and is able to find all exact nearest neighbors in 3.0–12.9x less time than the best alternative.

1.4 Outline

This thesis is organized as follows:

- In Chapter 2 we introduce our notation and formally define the problems addressed in the thesis, review some mathematics theory relevant to the following discussion, and present materials relevant to the experimental evaluation of our methods.
- In Chapter 3 we present an overview of prior work done on serial and parallel similarity search and nearest neighbor graph construction.
- In Chapter 4 we present our work on developing serial algorithms for cosine and Tanimoto ϵ-nearest neighbor graph construction.
- In Chapter 5 we present our work on efficient construction of k-nearest neighbor graphs using the cosine similarity function.
- In Chapter 6 we discuss high-performance shared memory parallel methods for both the ϵ and k-nearest neighbor graph construction problems.
- In Chapter 7, we discuss the collective impact of the works presented in this thesis and future research directions.

1.5 Related Publications

- **David C. Anastasiu** & George Karypis. L2AP: Fast Cosine Similarity Search With Prefix L-2 Norm Bounds. In *The 30th IEEE International Conference on Data Engineering* (ICDE 2014), pages 784-795, 2014.
- **David C. Anastasiu** & George Karypis. L2Knng: Fast Exact K-Nearest Neighbor Graph Construction with L2-Norm Pruning. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management* (CIKM '15), pages 791-800, ACM, 2015.
- **David C. Anastasiu** & George Karypis. PL2AP: Fast Parallel Cosine Similarity Search. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, in conjunction with SC'15 (IA3 2015), pages 1-8, ACM, 2015.

- **David C. Anastasiu** & George Karypis. Efficient Identification of Tanimoto Nearest Neighbors. Under submission: *IEEE International Conference on Data Science and Advanced Analytics* (DSAA'2016).
- **David C. Anastasiu** & George Karypis. Fast Parallel Cosine K-Nearest Neighbor Graph Construction. *To be submitted.*

Chapter 2

Background

In this chapter, we introduce our notation and formally define the problems addressed in the thesis, review some mathematics theory relevant to the following discussion, and present materials relevant to the experimental evaluation of our methods.

2.1 Definitions & Notation

Let $D = \{d_1, d_2, \ldots, d_n\}$ be a set of objects such that each object d_i is a (sparse) vector in an m dimensional feature space. We will use d_i to indicate the ith object, $\mathbf{d}_i$ to indicate the feature vector associated with the ith object, and $d_{i,j}$ to indicate the value (or weight) of the jth feature of object d_i.

Given object d_i, we denote by Γ_{d_i} its *neighborhood*, the set of objects in $D \setminus \{d_i\}$ with non-zero similarity with d_i, which are the *neighbors* of d_i. The number of objects in a neighborhood represents its size, which we denote by $|\Gamma_{d_i}|$. We focus on two related problems in this work. The ϵ-NNG construction problem seeks, for each object in D, all neighbors with a similarity value of at least ϵ, while the k-NNG construction problem seeks up to k neighbors with highest similarity values.

The similarity graph of D is a graph $G = (V, E)$ where vertices correspond to the objects and an edge (v_i, v_j) indicates that the jth object is in the neighborhood of the ith object and is associated with a weight, namely the similarity value $\text{sim}(d_i, d_j)$. We denote by G^ϵ the similarity graph containing edges for all neighbor pairs with similarity values of at least ϵ, and by G^k the similarity graph in which an edge exists between an

object and all its k-nearest neighbors. An *approximate* k-NNG $\tilde{G}^k$ is one in which the k neighbors of each vertex do not necessarily correspond to the k most similar objects. An *approximate* ϵ-NNG $\tilde{G}^\epsilon$ is one which does not have an edge for all neighbors of a vertex with similarity of at least ϵ, yet all similarities represented in the graph are at least ϵ. Note that the similarity values represented in all constructed graphs, either exact or approximate, are the exact values returned by the chosen similarity function.

For a given neighborhood Γ_{d_i}, we denote by the *minimum (neighborhood) similarity* σ_{d_i} the minimum similarity between object d_i and one of its current neighbors. We say that a *k-neighborhood* (a neighborhood in a k-NNG) is *improved* when its minimum similarity σ_{d_i} increases in value, and it is *complete* when adding any other neighbor to the k-neighborhood cannot increase σ_{d_i}. Similarly, we say that an *ϵ-neighborhood* is *improved* when a new neighbor with similarity at least ϵ is added to the neighborhood, and is *complete* when all remaining neighbors have similarities lower than ϵ.

The majority of feature values in sparse vectors are 0. As a result, a vector $\mathbf{d}_i$ is generally represented as the set of all pairs $(j, d_{i,j})$ satisfying $1 \leq j \leq m$ and $d_{i,j} > 0$. For a set of objects represented by sparse vectors, an *inverted index* representation of the set is made up of m lists, $\mathcal{I} = \{I_1, I_2, \ldots, I_m\}$, one for each feature. List I_j contains pairs $(d_i, d_{i,j})$, also called postings in the information retrieval literature, where d_i is an indexed object that has a non-zero value for feature j, and $d_{i,j}$ is that value. Postings may store additional statistics related to the feature within the document it is associated with.

Given a vector $\mathbf{d}_i$ and a dimension p, we will denote by $\mathbf{d}_i^{\leq p}$ the vector obtained by keeping the p leading dimensions in $\mathbf{d}_i$, $(d_{i,1}, \ldots, d_{i,p}, 0, \ldots, 0)$, which we call the (inclusive) *prefix* (vector) of $\mathbf{d}_i$. Similarly, we refer to $\mathbf{d}_i^{>p} = (0, \ldots, 0, d_{i,p+1}, \ldots, d_{i,m})$ as the (exclusive) *suffix* of $\mathbf{d}_i$, obtained by setting the first p dimensions of $\mathbf{d}_i$ to 0. The exclusive prefix $\mathbf{d}_i^{<p}$ and inclusive suffix $\mathbf{d}_i^{\geq p}$ are analogously defined.

Table 2.1 provides a summary of notation used in this thesis.

Table 2.1: Notation used throughout the work.

	Description
D	set of objects
d_i	the ith object
$\mathbf{d}_i$	vector representing ith object
$d_{i,j}$	value for jth feature in $\mathbf{d}_i$
$\mathbf{d}_i^{\leq p}, \mathbf{d}_i^{>p}$	(inclusive) prefix and (exclusive) suffix of $\mathbf{d}_i$ at dimension p
$\mathbf{d}_i^{<p}, \mathbf{d}_i^{\geq p}$	exclusive prefix and inclusive suffix of $\mathbf{d}_i$ at dimension p
$\mathbf{d}_i^{\leq}, \mathbf{d}_i^{>}$	un-indexed/indexed portion of $\mathbf{d}_i$
$\hat{\mathbf{d}}_i$	normalized version of $\mathbf{d}_i$
σ_{d_i}	smallest similarity value in Γ_{d_i}
$\mathcal{I}$	inverted index
$\mathbf{f}_j$	vector containing jth feature values from all objects
Γ_{d_i}	neighborhood for object d_i
$\mathcal{N}$	set of neighborhoods
$\hat{\mathcal{N}}$	set of initial approximate neighborhoods
ϵ	minimum desired similarity
k	size of desired neighborhoods
μ	candidate list size
γ	number of neighborhood enhancement updates
δ	early neighborhood enhancement termination
ν	number of completion blocks
ζ	number of non-zeros in an inverted index tile
η	number of objects in a query tile
nt	number of threads

2.2 Theory Background

For any two vectors $\mathbf{d}_i$ and $\mathbf{d}_j$ in $\mathcal{R}^m$, the *standard inner product*, or *dot-product*, between them is

$$\left\langle \mathbf{d}_i, \mathbf{d}_j \right\rangle = \mathbf{d}_i^T \mathbf{d}_j = \sum_{l=1}^{m} d_{i,l} d_{j,l} \ , \tag{2.1}$$

where T denotes the vector transpose. In this work, we will prefer the $\left\langle \mathbf{d}_i, \mathbf{d}_j \right\rangle$ notation for the dot-product.

The *norm* of a vector $\mathbf{d}$ is a function $f : \mathcal{R}^m \rightarrow \mathcal{R}^+$, which measures the *length* of the vector. A vector norm satisfies the following properties:

- positive definiteness: $f(\mathbf{d}) \geq 0$ for all $\mathbf{d} \in \mathcal{R}^m$ and $f(\mathbf{d}) = 0$ iff $\mathbf{d} = 0$,
- absolute homogeneity: $f(\alpha \mathbf{d}) = |\alpha| \mathbf{d}$ for any $\alpha \in \mathcal{R}$,

- triangle inequality: $f(\mathbf{d}_i + \mathbf{d}_j) \leq f(\mathbf{d}_i) + f(\mathbf{d}_j)$, for all $\mathbf{d}_i,\ \mathbf{d}_j \in \mathcal{R}^m$.

Some popular vector norms referenced throughout this work include the *Chebyshev* or ℓ_∞-norm, computed as

$$\|\mathbf{d}_i\|_\infty = \max\{d_{i,1}, d_{i,2}, \ldots, d_{i,m}\}, \tag{2.2}$$

and the ℓ_p-norm,

$$\|\mathbf{d}_i\|_p = (|d_{i,1}|^p + |d_{i,2}|^p + \ldots + |d_{i,m}|^p)^{1/p}. \tag{2.3}$$

The *distance* between two vectors is measured as the norm of their difference, $\|\mathbf{d}_i - \mathbf{d}_j\|$. The generic ℓ_p-norm based distance, $\|\mathbf{d}_i - \mathbf{d}_j\|_p$, is called the Minkowski distance. Two popular versions of the Minkowski distance are the Manhattan or Taxicab distance (ℓ_1-norm) and the Euclidean distance (ℓ_2-norm). The ℓ_2-norm is also often generically referred to in literature as the vector *magnitude*, or *length*.

$$\|\mathbf{d}_i\|_1 = |d_{i,1}| + |d_{i,2}| + \ldots + |d_{i,m}| \qquad (\ell_1\text{-norm}), \tag{2.4}$$

$$\|\mathbf{d}_i\|_2 = (d_{i,1}^2 + d_{i,2}^2 + \ldots + d_{i,m}^2)^{1/2} = \sqrt{\langle \mathbf{d}_i, \mathbf{d}_i \rangle} \qquad (\ell_2\text{-norm}). \tag{2.5}$$

The Hölder inequality provides an upper bound for the dot-product of two vectors based on their length. Specifically, given scalar values p and q such that $1/p + 1/q = 1$, it states that

$$|\langle \mathbf{d}_i, \mathbf{d}_j \rangle| \leq \|\mathbf{d}_i\|_p \|\mathbf{d}_j\|_q. \tag{2.6}$$

For the norms previously defined, it follows that

$$|\langle \mathbf{d}_i, \mathbf{d}_j \rangle| \leq \|\mathbf{d}_i\|_1 \|\mathbf{d}_j\|_\infty, \tag{2.7}$$

$$|\langle \mathbf{d}_i, \mathbf{d}_j \rangle| \leq \|\mathbf{d}_i\|_\infty \|\mathbf{d}_j\|_1, \tag{2.8}$$

$$|\langle \mathbf{d}_i, \mathbf{d}_j \rangle| \leq \|\mathbf{d}_i\|_2 \|\mathbf{d}_j\|_2. \tag{2.9}$$

Proposition 2.9 is also known as the Cauchy-Schwarz inequality, and can be equivalently written as

$$\langle \mathbf{d}_i, \mathbf{d}_j \rangle^2 \leq \langle \mathbf{d}_i, \mathbf{d}_i \rangle \langle \mathbf{d}_j, \mathbf{d}_j \rangle. \tag{2.10}$$

The following additional inequalities between different norms can also be proved,

$$\|\mathbf{d}\|_\infty \leq \ \|\mathbf{d}\|_1 \leq m\|\mathbf{d}\|_\infty \tag{2.11}$$

$$\|\mathbf{d}\|_\infty \leq \ \|\mathbf{d}\|_2 \leq \sqrt{m}\|\mathbf{d}\|_\infty \tag{2.12}$$

$$\|\mathbf{d}\|_2 \leq \|\mathbf{d}\|_\infty \leq \sqrt{m}\|\mathbf{d}\|_2. \tag{2.13}$$

The interested reader may find proofs for many of the properties described in this section in the excellent matrix computations reference by Golub and Van Loan [24].

The ℓ_0-*pseudo-norm* represents the function that counts the number of non-zero values in a vector. Denoted as $\|\mathbf{d}\|_0$, the ℓ_0-norm of $\mathbf{d}$ is not a true vector norm, as it does not satisfy the absolute homogeneity property. Given the prefix and suffix vector definitions given in Section 2.1, one can then verify, for a given prefix feature p, that

$$\mathbf{d}_i = \mathbf{d}_i^{\leq p} + \mathbf{d}_i^{>p}, \tag{2.14}$$

$$\|\mathbf{d}_i\|_0 = \|\mathbf{d}_i^{\leq p}\|_0 + \|\mathbf{d}_i^{>p}\|_0, \tag{2.15}$$

$$\|\mathbf{d}_i\|_1 = \|\mathbf{d}_i^{\leq p}\|_1 + \|\mathbf{d}_i^{>p}\|_1, \tag{2.16}$$

$$\|\mathbf{d}_i\|_2^2 = \|\mathbf{d}_i^{\leq p}\|_2^2 + \|\mathbf{d}_i^{>p}\|_2^2, \tag{2.17}$$

$$\|\mathbf{d}_i\|_\infty = \max(\|\mathbf{d}_i^{\leq p}\|_\infty, \ \|\mathbf{d}_i^{>p}\|_\infty), \text{ and} \tag{2.18}$$

$$\langle \mathbf{d}_i, \mathbf{d}_j \rangle = \langle \mathbf{d}_i, \mathbf{d}_j^{\leq p} \rangle + \langle \mathbf{d}_i, \mathbf{d}_j^{>p} \rangle. \tag{2.19}$$

These properties are evident from the presented definitions and the fact that the set of non-zero values in the vector is the union of the disjoint sets of non-zero values in the prefix and suffix vectors.

Given a vector norm $\|\cdot\|$, a vector can be scaled to have unit norm, an operation called *normalization*, by dividing it by its length with respect to that norm. Vector normalization changes the length of the vector without changing its direction. A number of methods presented in this work make use of ℓ_2-norm normalized vectors, denoted as

$$\hat{\mathbf{d}} = \frac{\mathbf{d}}{\|\mathbf{d}\|_2}. \tag{2.20}$$

When clear from context, we will drop the hat from the notation and denote the normalized version of the ith object vector representation as $\mathbf{d}_i$. Specifically, in our description

of our cosine similarity based nearest neighbor graph construction methods, we assume input vectors have been normalized and denote the normalized version of the ith object vector representation as $\mathbf{d}_i$.

2.2.1 Similarity Functions

While there have been many proposed similarity functions between weighted vectors, we focus on a subset of popular similarity measures that take advantage of sparsity when comparing objects. In other words, the similarity function can be computed by traversing only the non-zero values in the two vectors.

$$\textit{Cosine similarity}: \qquad \mathrm{C}(\mathbf{d}_i, \mathbf{d}_j) = \frac{\langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\|_2 \|\mathbf{d}_j\|_2} = \langle \hat{\mathbf{d}}_i, \hat{\mathbf{d}}_j \rangle \tag{2.21}$$

$$\textit{Tanimoto similarity}: \qquad \mathrm{T}(\mathbf{d}_i, \mathbf{d}_j) = \frac{\langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\|_2^2 + \|\mathbf{d}_j\|_2^2 - \langle \mathbf{d}_i, \mathbf{d}_j \rangle} \tag{2.22}$$

Tanimoto similarity is also known in the literature as the Tanimoto coefficient or the (extended) Jaccard coefficient.

Given non-negative real-valued vectors, which represent the type of input our methods are designed for, the following properties hold for these similarity measures.

- Similarity values range in $[0, 1]$, and $\mathrm{sim}(\mathbf{d}_i, \mathbf{d}_j) = 1$ only if $\mathbf{d}_i = \mathbf{d}_j$, where $\mathrm{sim}(\cdot, \cdot)$ represents any of the similarity measures above.
- Symmetry: $\mathrm{sim}(\mathbf{d}_i, \mathbf{d}_j) = \mathrm{sim}(\mathbf{d}_j, \mathbf{d}_i)$.

Another popular proximity measure is the Euclidean distance, computed as the ℓ_2-norm of the difference between the two vectors,

$$\mathrm{E}(\mathbf{d}_i, \mathbf{d}_j) = \|\mathbf{d}_i - \mathbf{d}_j\|_2. \tag{2.23}$$

Dissimilarity measures, such as the Euclidean distance, can be converted into a measure of similarity via a monotonic decreasing function. For example,

$$\mathrm{ES}(\mathbf{d}_i, \mathbf{d}_j) = e^{-\|\mathbf{d}_i - \mathbf{d}_j\|_2^2},$$

is a popular transformation that relates the squared loss function to the negative log-likelihood of group membership in clustering [25]. Euclidean distance is symmetric, and, since it is a norm, its values are positive definite, absolute homogeneous, and follow triangle inequality,

$$\mathrm{E}(\mathbf{d}_i, \mathbf{d}_k) \leq \mathrm{E}(\mathbf{d}_i, \mathbf{d}_j) + \mathrm{E}(\mathbf{d}_j, \mathbf{d}_k). \tag{2.24}$$

2.3 Datasets

We use a diverse set of datasets from three different domains to evaluate the performance of our methods, namely text documents, networks (graph), and chemical compounds. They represent some benchmark text corpora popularly used in text-categorization research and several real-world web/social networks and chemical compound datasets. We chose these datasets because they represent real-world problems, yet are quite varied with respect to their number of objects and row and column size. Below, we give additional details about each dataset. Their characteristics, including number of objects (n), features (m), millions of non-zeros (nnz), and row/column mean number of non-zeros are detailed in Table 2.2.

Text datasets:

- **RCV1** is a standard benchmark corpus containing over 800,000 newswire stories provided by Reuters, Ltd. for research purposes, We use version 2 of the dataset, also known as RCV1-v2, made available by Lewis et al. [26].
- **RCV1-400k** and **RCV1-100k** are random subsets of 400,000 and 100,000 documents, respectively, from RCV1.
- **WW-500k** was kindly provided to the authors by Satuluri and Parthasarathy [27], along with the WW-100k and Wiki datasets. It contains documents with at least 200 distinct features, extracted from the September 2010 article dump of the English Wikipedia.
- **WW-100k** contains documents from the WW-500k dataset with at least 500 distinct features.
- **WW200** contains documents with at least 200 distinct features, extracted by the

author from the October 2014 article dump of the English Wikipedia[1].

- **WW500** contains the subset of documents from WW200 with at least 500 distinct features.
- **WW200-250k** is a random subset of 250,000 objects from WW200.
- **Patents** is a random subset of 100,000 patent documents from all US utility patents[2] . Each document contains the patent title, abstract, and body.

Network datasets:

- **Twitter**, first provided by Kwak et al. [28], contains *follow* relationships of a subset of Twitter users that follow at least 1,000 other users. Vectors represent users, and features are users they follow.
- **Wiki** represents a directed graph of hyperlinks between Wikipedia articles in the Wiki dump.
- **Orkut** contains the friendship network of over 3M users of the Orkut social media site, made available by Mislove et al. [29]. Vectors represent users, and features are friends of the users.

Chemical datasets:

- **MLSMR** [30] (Molecular Libraries Small Molecule Repository) is a collection of structures of compounds accepted into the repository of PubChem, NCBI's database of small organic molecules and their biological activity. We used the December 2008 version of the SDF database[3].
- **SC** contains chemical compounds from the SureChEMBL [31] database, which includes a large set of compounds automatically extracted from text, images and attachments of patent documents.
- **SC-5M**, **SC-1M**, **SC-500k** and **SC-100k** are random subsets of 5,000,000, 1,000,000, 500,000 and 100,000 compounds, respectively, from the **SC** dataset.

[1] http://download.wikimedia.org
[2] http://www.uspto.gov/
[3] https://mlsmr.evotec.com/MLSMR_HomePage/pdf/MLSMR_Collection_20081201.zip

Table 2.2: Dataset statistics.

dataset	**type**	n	m	$nnz \cdot 10^6$	μ_r	μ_c
RCV1	text	804,414	45,669	62	77	1,348
RCV1-400k	text	400,000	45,669	31	77	670
RCV1-100k	text	100,000	45,669	8	78	187
WW-500k	text	494,244	343,622	197	399	574
WW-100k	text	100,528	339,944	79	787	233
WW200	text	1,017,531	663,419	437	430	659
WW500	text	243,223	660,600	202	830	306
WW200-250k	text	250,000	663,410	108	430	164
Patents	text	100,000	759,044	46	464	61
Twitter	network	146,170	143,469	200	1370	1395
Wiki (`L2AP`)	network	1,815,914	1,648,879	44	24	27
Wiki (`pL2AP`)	network	3,714,404	3,714,401	111	30	56
Orkut	network	3,072,626	3,072,441	223	73	73
MLSMR	chemical	325,164	20,021	56	173	2,803
SC	chemical	11,519,370	7,415	1,785	155	262,669
SC-5M	chemical	5,000,000	7,415	700	155	103,063
SC-1M	chemical	1,000,000	6,752	155	155	22,949
SC-500k	chemical	500,000	6,717	78	155	11,533
SC-100k	chemical	100,000	6,623	16	155	2,336

In the table, n represents the number of objects (rows), m is the number of features in the vector representation of the objects (columns), nnz is the number of non-zero values (measured in millions), and μ_r and μ_c are the mean number of non-zeros in each row and column, respectively.

2.4 Data Processing

2.4.1 Text Data Processing

We use standard text processing methods to encode documents as sparse vectors. Each document is first tokenized, removing punctuation, making text lower-cased, and splitting the document into a set of words. Each word is stemmed using the Porter stemmer [32], reducing different versions of the same word to a common token. Within the space of all tokens, a document is then represented by the sparse vector containing the frequency of each token present in the document. When computing cosine nearest neighbor graphs, as is customary in text analysis tasks, we scaled frequency based vectors by the inverse document frequency [16], which reduces the importance of terms frequently used in the corpus when computing similarities.

2.4.2 Network Data Processing

We represent relationships for a given object in a network as a sparse binary vector within the space of the number of objects in the network. Each value in the vector denotes whether a relationship (edge) exists between the object represented by the vector and another object. In general, the relationships are not symmetric. A vector representing an object can be thought of as a row or column in the adjacency matrix representing the network (graph), depending on the type of relationship (in-coming or outgoing edges). When computing cosine nearest neighbor graphs, we scaled binary vectors by the inverse document frequency.

2.4.3 Chemical Compound Processing

We encode each chemical compound as a sparse frequency vector of the molecular fragments it contains, represented by GF [33] descriptors extracted using the AFGen v. 2.0 [34] program[4]. AFGen represents molecules as graphs, with vertices corresponding to atoms and edges to bonds in the molecule. GF descriptors are the complete set of unique size-bounded subgraphs present in each compound. Within the space of all GF descriptors for a compound dataset, a compound is then represented by the sparse vector containing the frequency of each GF descriptor present in the compound. We used a minimum length of 3 and a maximum length of 5 and ignored Hydrogen atoms when generating GF descriptors (AFGen settings *fragtype=GF, lmin=3, lmax=5, fmin=1, noh: yes*). Before running AFGen on each chemical dataset, we used the Open Babel toolbox [35] to remove compounds with incomplete descriptions.

2.5 Performance Measures

When comparing the search performance of different methods, an important characteristic in our experiments is CPU runtime, which is measured in seconds. I/O time needed to load the dataset into memory or write output to the file system should be the same for all methods and is, in general, ignored[5]. Between a method A and a baseline

[4] http://glaros.dtc.umn.edu/gkhome/afgen/download

[5] Experiments for our `L2AP` and `L2Knng` prototypes (see Sections 4.4 and 5.2) and all related baselines measured the total execution time, including I/O.

method B, we report speedup as the ratio of B's execution time and that of A's.

We use average recall to measure the accuracy of the returned result when comparing the performance of approximate graph construction methods. We obtain the true neighborhood graph via brute-force search, then compute the average recall as,

$$R = \frac{1}{|D|} \sum_{d_i \in D} \frac{\text{\# true neighbors in } N_{d_i}}{|N_{d_i}|}.$$

We follow others in using the number of full similarity computations as an architecture and programming language independent way to measure nearest neighbor graph construction cost [36, 37]. However, we use a slightly different normalization constant, $NC = n(n-1)$, as some of our baselines may not take advantage of symmetry in similarity computations, and thus may compute up to $n-1$ similarity values for each vector in the dataset. We report, for all algorithms, *scan rate* = *# similarity evaluations*/NC, and *candidate rate* = *# candidates*/NC.

We report the performance of parallel methods in two ways. We compare the efficiency of all parallel methods against the best existing serial algorithm for solving the problem and report execution times and/or speedup values. Additionally, we report strong scaling results, in which multi-threaded execution times for a parallel method are compared with the 1-threaded execution of the same method.

2.6 Execution Environment

Depending on the resources available at the time of our completing a research prototype, we used several environments for executing our experimental evaluations. For each prototype, we executed all experiments for our methods and all related baselines in the same environment. In this section, we describe the environments we used and parameters we chose for our methods and baselines, the algorithms for which will be presented later in this thesis.

Cosine ϵ-NNG Construction

Our method, `L2AP`, and all baselines for cosine ϵ-NNG construction are single-threaded, serial programs, implemented in C and compiled using gcc 4.4.6 with the

-O3 optimization setting enabled. The `BayesLSH` package[6] includes implementations for `LSH`, `AllPairs` +`BayesLSH-Lite`, `LSH` +`BayesLSH-Lite`, and `AllPairs`. An implementation of `MMJoin` was not available. We implemented `IdxJoin`, `AllPairs`[7], `MMJoin`, `L2AP`, and `L2AP-a`[8] in our prototype. Each method was executed on its own node in a cluster of HP ProLiant BL280c G6 blade servers, each with 2.8 GHz Intel Xeon processors and 24 Gb RAM. Methods that took longer than 48 hours to execute were terminated. For each method, we varied the similarity threshold between 0.3 and 0.95, in increments of 0.05. To further qualify the utility of our method for near-duplicate object detection, we also executed each method for similarities between 0.96 and 0.99, in increments of 0.01. As suggested by Satuluri and Parthasarathy, we used $r = 0.03$ (97% recall) and checked $h = 128$ hashes in both `BayesLSH-Lite` and `L2AP-a` approximate pruning. For approximate methods, we executed each test a minimum of three times and report the average time over all test executions.

Tanimoto ϵ-NNG construction

Our method, `TAPNN`, and all baselines for Tanimoto ϵ-NNG construction are single-threaded, serial programs, implemented in C and compiled using gcc 5.1.0 with the -O3 optimization setting enabled. Each method was executed on its own node in a cluster of HP Linux servers. Each server is a dual-socket machine, equipped with 24 Gb RAM and two four-core 2.6 GHz Intel Xeon 5560 (Nehalem EP) processors with 8 Mb Cache. We executed each method a minimum of four times for $\epsilon \in \{0.6, 0.7, 0.8, 0.9, 0.99\}$ and report the best execution time in each case. Due to its size (14 Gb), we executed data scaling experiments involving the full SC dataset on a different server, equipped with 64 Gb RAM and two 12-core 2.5 GHz Intel Xeon (Haswell E5-2680v3) processors with 30 Mb Cache. As all tested methods are serial, only one core was used on each server during the execution.

Cosine k-NNG construction

[6] See http://www.cse.ohio-state.edu/~satuluri/research.html

[7] Unlike the `AllPairs` mplementations by Bayardo et al. and in the `BayesLSH` package, ours uses a dense representation of the query vector. We found our implementation, on average, to be 2.5x faster than the one in the `BayesLSH` package and use it as baseline in this work. This implementation detail has since been incorporated into the other packages.

[8] Source code for all methods is available at http://cs.umn.edu/~dragos/l2ap

Our method, `L2Knng`, and all baselines for cosine k-NNG construction are single-threaded, serial programs. A C++ based library implementing *NN-Descent* can be found at http://www.kgraph.org/. We implemented[9] `kIdxJoin`, `kL2AP`, *Greedy Filtering*[10], *Maxscore*, *BMM*, `L2Knng`, and `L2Knng-a` in C and compiled our program using gcc 4.4.7 with -O3 optimization. Each method was executed on its own node in a cluster of HP ProLiant BL280c G6 blade servers, each with 2.8 GHz Intel Xeon processors and 24 Gb RAM.

We executed each method for $k \in \{1, 5, 10, 25, 50, 75, 100\}$ and tuned parameters to achieve balanced high recall and efficient execution. For all `L2Knng` and `L2Knng-a` experiments, we set the parameter $\delta = 0.0001$. We tested `kL2AP` by decreasing the threshold t in steps of 0.1, 0.25, and 0.5, and report the best results among the step choices. For the *NN-Descent* library[11], we set $\rho = 1$, $S = 20$, and indexing $K = \mu$ (the candidate list size $\mu \geq k$). For all stochastic methods, we executed a minimum of 5 runs for each set of parameter values and we report averages of all recorded times.

Parallel cosine ϵ-NNG construction

Our method, `pL2AP`, and all baselines are implemented in C and compiled using gcc 4.4.7 with -O3 optimization. We used the OpenMP framework for implementing shared-memory parallel methods. Each method was executed on its own node in a cluster of HP Linux servers. Each server is a dual-socket machine, equiped with 64 Gb RAM and two twelve-core 2.5 GHz Intel Xeon E5-2680v3 processors with 30 Mb Cache. For each method, we varied the similarity threshold ϵ between 0.3 and 0.9, in increments of 0.1. For `pL2AP`, we fixed η at $25K$ objects and varied ζ between $250K$ and $4M$ in $250K$ increments. We set the masked hash-table size parameter h to 2^{13}.

Parallel cosine k-NNG construction

Our method, `pL2Knng`, and all baselines are implemented in C and compiled using gcc 5.1.0 with the -O3 optimization setting enabled. We used the OpenMP framework for implementing shared-memory parallel methods. Each method was executed on its

[9]Source code available at http://cs.umn.edu/~dragos/l2knng.

[10]The authors of *Greedy Filtering* kindly provided a Java-based implementation of their algorithm for comparison. On average, our C implementation achieved 1.13x speedup over the Java one.

[11]We thank Wei Dong for his invaluable assistance with using the KGraph library and finding *NN-Descent* evaluation parameters.

own node in a cluster of HP Linux servers. Each server is a dual-socket machine, equipped with 64 Gb RAM and two eight-core 2.6 GHz Intel Xeon E5-2670 (Sandy Bridge) processors with 20 Mb Cache.

We executed each method for

$$k \in \{10, 25, 50, 75, 100, 200, 300, 400, 500\}$$

and tuned parameters for each method to achieve balanced high recall and efficient execution. For all `L2Knng` based methods, we set the parameter $\delta = 0.0001$. We used the latest version of the *NN-Descent*[12] library available at the time of our experiments (v.1.4), and set $\rho = 1$, and indexing $K = \mu$ (the candidate list size $\mu \geq k$). For all stochastic methods, we executed a minimum of 3 runs for each set of parameter values and we report averages of all recorded times.

[12] http://www.kgraph.org/releases/kgraph-1.4-x86_64.tar.gz

Chapter 3

Related Work

In this section, we will give an overview of some important works addressing our two nearest neighbors graph construction problems of interest. We will focus our discussion first on methods for cosine nearest neighbor graph construction, and then discuss existing extensions for the Tanimoto similarity.

3.1 ϵ-NNG Construction

The ϵ-NNG construction problem has its roots in the *similarity join* problem from the database community [5,38]. In that context, Chaudhuri et al. first formalized the *prefix-filtering* principle [38], showing that only a few elements from the beginning of a query vector must be checked against other vectors to find all necessary candidates. Bayardo et al. [8] disconnected the problem from the underlying database system and developed additional pruning strategies based on a predefined vector order in the dataset. They also introduced *dynamic indexing*, leveraging the prefix-filtering principle to index only a portion of each vector after its candidate list was generated.

The majority of subsequently developed ϵ-NNG construction methods follow the same format as in the method described by Bayardo et al. They proceed in three stages. First, during *candidate generation*, a list of objects is compiled whose similarity scores to the query object are believed to exceed the threshold. Potential candidates during this stage are vetted based on different theoretic upper bounds on the similarity. The *candidate verification* stage finalizes the similarity computation for identified candidates

and compares it against the threshold ϵ. Additional pruning may reduce the number of full similarities being computed. Finally, the query object is indexed (*indexing* stage) before continuing with the next object in the processing order.

A number of extensions have focused on the set-based or binary representation of objects. While not directly applicable to our domain, they provide insights into types of pruning that may be beneficial during the graph construction. Xiao et al. [6] first introduced a tighter indexing bound and *positional filtering* in `PPJoin`. During candidate generation, positional filtering provides additional pruning based on the remaining size of the vectors once a feature is found in common. Motivated by experimental results showing quadratic growth in the candidate pool size in `AllPairs`, Xiao et al. pushed filtering into the candidate verification stage through Hamming distance based *suffix filtering*. When considering string similarity search using the edit distance measure, Xiao et al. [39] showed that the problem can be efficiently solved using q-gram-based mismatch filtering. Xiao et al. [40] introduced further `AllPairs` optimizations to answer top-k queries efficiently. While previous algorithms focused on reducing the generated candidate pool size, Ribeiro and Härder [41] sought to reduce overall search time by minimizing the size of the inverted index through dynamic min-prefix indexing. They coupled a cheap candidate generation step with additional stopping criteria in the verification stage to improve on `AllPairs` and `PPJoin`. Wang et al. [42] sought to reach a balance between the number of candidates being generated and the number of pruned candidates, which lead them to develop of a cost-based scheme for choosing variable-length prefixes.

There has been little focus, in comparison, on solving the ϵ-NNG construction problem for weighted vectors and cosine similarity. Bayardo et al. [8] gave the first integrated solution for the problem, the `AllPairs` algorithm. In `APT`, Awekar and Samatova [43] provided tighter bounds over `AllPairs` on the candidate vector minimum size and similarity score estimate. Lee et al. [15] introduced *length filtering* and length-based *suffix filtering* in `MMJoin`. As they are pertinent to our problem, we detail these methods in Section 4.1. We then show, both theoretically and experimentally, that our pruning strategies outperform those in these methods.

Although emphasis has recently shifted to solving ϵ-NNG construction exactly, approximate methods remain popular, especially in domains only interested in objects with

high similarity thresholds. In the context of near-duplicate object detection, Broder et al. [1] applied similarity search to sketches built using min-wise independent permutations of shingled Web documents. A popular alternative, Locality Sensitive Hashing (LSH) [44, 45], uses families of functions that hash similar objects to the same bucket with high probability to generate candidate sets. Zhai et al. [46] presented a probabilistic algorithm for similarity search based on random filters. Note that these algorithms solve a related problem, that of *nearest neighbor(s) search*, in which the query objects are not assumed to be a part of the input set. The ϵ-NNG could be constructed by executing a nearest neighbors query for each of the objects in the input set, but its construction does not consider the computation structure inherent in the ϵ-NNG construction problem. To take advantage of this structure, Satuluri and Parthasarathy [27] introduced `BayesLSH`, a principled Bayesian approach for candidate pruning and similarity estimation, which they combine with candidate generation steps from `AllPairs` and `LSH`.

3.1.1 Tanimoto ϵ-NNG Construction

Within the chemoinformatics community, a great deal of effort has been spent trying to accelerate pairwise similarity computations using the Tanimoto coefficient. Swamidass and Baldi [47] described a number of *bounds* for fast exact threshold based Tanimoto similarity searches of binary and integer based vector representations of chemical compounds. These bounds allow skipping many object comparisons that will theoretically not be similar enough to be included in the result. Baldi et al. [48] proposed an algorithm for pruning the similarity search space through exclusive OR (XOR) operations on compressed bit-vector representations of the molecules. Nasr et al. [49] developed hashing techniques for pruning the search space by transforming bounds on the intersection of molecule hash signatures to the Tanimoto similarity of their fingerprint vectors. Kristensen et al. [50] and Smellie [51] relied instead on tree-based data structures to speed up similarity search. Tabei and Tsuda [52] described *SketchSort*, an approximate method which uses min-wise independent permutation based locality sensitive hashing to solve the ϵ-NNG construction problem. Most recent approaches focus on speeding up chemical searches using inverted index data structures borrowed from information retrieval [49, 53, 54]. Among all existing methods, algorithms taking advantage of inverted

indexes have been shown to be the most effective means for identifying neighbors when dealing with objects represented in a high-dimensional space.

The numeric representation of chemical compounds is still an open problem in chemoinformatics. Due to computation efficiency constraints, initial representations focused on capturing the presence or absence of features within the compound. Compounds were represented as binary vectors, referred to as a *fingerprints*. In recent years, frequency (or counting) vectors, which capture how many times a feature is present, and real valued vectors, called *descriptors*, have gained popularity [9,55]. Arif et al. [56], for example, investigated the use of inverse frequency weighting of features in frequency descriptors for similarity-based Virtual Screening, and found marked increases in screening effectiveness in some circumstances. The efficient Tanimoto ϵ-NNG construction methods described in this thesis may be used to compute bounded pairwise similarities of descriptor vectors, accelerating chemical compound search and other Virtual Screening tasks.

Tanimoto similarity has also been widely used in the computer science community. As noted earlier, a number of data mining methods have been devised for solving the ϵ-NNG construction problem. While most of the existing work addresses either binary vector object representations [6, 40, 57] or cosine similarity [18, 43], Bayardo et al. [8] and Lee et al. [15] showed how their filtering based ϵ-NNG construction methods can be extended to the Tanimoto coefficient for binary and real-valued vectors, respectively. Focusing on real-valued vectors, Kryszkiewicz [20, 58] proved several theoretic bounds on the Tanimoto similarity and sketched an inverted index based algorithm for efficient similarity search.

3.2 k-NNG Construction

Relatively few k-NNG construction algorithms have been designed to address cosine similarity. Park et al. [37] described *Greedy Filtering*, an approximate filtering-based approach which prioritizes computing similarities between objects with high weight features in common. After first reordering the dimensions of each vector based on their weight, in decreasing weight order, the algorithm builds a partial inverted index, which it uses to find candidates for each object. Candidates for an object d_i are those objects

in the inverted index lists associated with the leading dimensions in $\mathbf{d}_i$, i.e., the prefix of $\mathbf{d}_i$. *Greedy Filtering* indexes enough of each vector's prefix as to lead to at least μ candidates for each object. After all prefixes are identified and the partial inverted index is constructed, *Greedy Filtering* computes pairwise similarities of objects in each inverted index list, which can lead to much more than μ similarity computations for each object, and repeated computations for pairs of objects with two or more common features in their prefixes.

In *NN-Descent*, Dong et al. [36] followed an iterative neighborhood improvement strategy based on the intuition that similar objects are likely to be found among the neighborhoods of objects in a query object's neighborhood. Starting with a randomly chosen initial k-NNG, their method iteratively improves the graph by computing, for each object d_i, via a *local join*, pairwise similarities between d_i, objects in its neighborhood, and those objects that contain d_i in their neighborhoods. The neighborhoods of both objects participating in a similarity computation are updated with the result. The method avoids duplication of effort between iterations by only allowing an object to participate in the local join if it has been added to some neighborhood in the last update. Sampling and early termination parameters provide a way to control the compromise between algorithm runtime and recall. However, *NN-Descent* computes $O(nk^2)$ object similarities in its first iteration. Furthermore, the algorithm does not provide a way to filter out candidates that are unlikely to improve the query object's neighborhood.

Top-k document retrieval is a related problem from information retrieval, which has had many proposed solutions over the years. Most methods in this class have been designed for very large document collections, and have focused on minimizing and/or parallelizing operations needed to quickly answer fairly short input queries. Result sets are in most cases inexact. Some recent works use an in-memory inverted index and pruning, called *safe early termination*, to return the same result set as an exhaustive search [59–63]. One could then solve the exact k-NNG problem by executing n top-k queries with one of these methods, one for each of the input objects. In their Block-Max WAND (*BMW*) method [61], Ding and Suel use an augmented index structure, called a Block-Max index, which stores inverted lists as compressed blocks of postings, along with the maximum score that could be achieved given the values in the block postings. By using the block maximum scores for early termination, many blocks can be skipped,

resulting in improved execution. Dimopoulos et al. [62] extended the work of Ding and Suel and designed several methods that take advantage of Block-Max type indexes. Among them, docID-oriented Block-Max Maxscore with variable block sizes (*BMM*) has been shown to outperform the others and several baselines (including *BMW*) for long queries. The method partitions the postings in each inverted list into blocks of equally-sized ID ranges, allowing fast look-up for the block a document's posting may be found in. Block sizes vary based on the number of postings in each list. For each block, *BMM* also keeps track of the maximum document ID and maximum score for any of the postings in the block. The *Maxscore* [64] algorithm described by Turtle and Flood was then adapted to use block maximum scores for early termination.

Locality Sensitive Hashing (LSH) [44, 45] uses families of functions that hash *signatures* of similar objects to the same bucket with high probability. The objects in the buckets that a query object hashes to can be considered its neighbors. The similarity with a neighbor can then either be estimated by comparing the object signatures or computed exactly. Created initially to solve the top-k retrieval problem, LSH has been shown effective at solving the nearest-neighbor problem (1-NNG), but suffers from low recall as the required neighborhood size increases [61]. Some recent LSH variations have tackled the k-NNG problem specifically (e.g., E2LSH [65] and DSH [66]), but focus on distance functions between objects, such as the Euclidean distance.

While ϵ-NNG construction algorithms cannot be used directly to construct a k-NNG, as we do not know the appropriate threshold that will lead to generating the complete k-NNG, some of the techniques used in ϵ-NNG construction algorithms can be adapted to prune the search space when solving the k-NNG construction problem. We provide an overview of existing ϵ-NNG construction methods in Section 3.1.

A number of k-NNG construction algorithms have been proposed for *metric spaces*, where we seek the k objects with the smallest metric distance from the query. Tree-based data structures are often used to facilitate partitioning the search space, allowing neighbor searches to be prioritized within grids close to the one the query object is in [67]. These types of methods have been shown effective in low dimensional spaces, but do not scale well as dimensionality increases.

3.3 Parallel Algorithms

Existing distributed solutions for nearest neighbor graph construction generally use the MapReduce [68] framework and can be thought of as belonging to one of two categories. Most rely on the framework's built-in features to aggregate (reduce) partial similarities of object pairs computed in mappers [69–72]. The computation efficiency can be greatly increased by first generating an inverted index for the set of objects, which can be done using one MapReduce task. The postings in the inverted index lists can then be combined with features in the object vectors or with other postings in the same list to generate partial similarity scores. While some pruning strategies can be used to avoid generating some partial scores, these methods often suffer from high communication costs which make then inefficient for large datasets [73].

The second category of MapReduce methods use a mapper-only scheme, with no reducers [73–75]. They partition the set of objects into subsets (blocks) and use serial ϵ-NNG construction methods to find pairwise similarities of objects in block pairs. Certain block comparisons can be eliminated by relying on block-level filtering techniques, such as computing the similarity of the objects made up of the maximum values for features in the two blocks. When comparing two blocks, Alabduljalil et al. proposed locally building a full inverted index for one of the blocks and scanning through query objects in the other block to compute their similarity. They found that filtering candidates was detrimental to execution speed and suggested removing this optimization, rendering their local search identical to that performed in one tile by our naïve baseline, `pIdxJoin`. Within this context, they examined distributed load balancing strategies [75] and cache-conscious performance optimizations for the local searches [74]. They provided a cost based analysis aimed at finding sizes for comparison blocks that maximize cache locality. Their analysis was based on a full inverted index and mean lengths of vectors and inverted lists, which can vary greatly in real datasets.

Existing multi-core cosine ϵ-NNG construction solutions are limited to the parallel `APT` (`pAPT`) algorithm by Awekar and Samatova [76]. After first indexing all input objects, `pAPT` allows threads to share the data structure during the candidate generation and verification stages, which use the same pruning strategies Awekar and Samatova proposed in `APT` [43]. Some of the pruning leads to reducing the size of the inverted

index by advancing pointers for the beginning of inverted lists. Awekar and Samatova avoided a costly synchronization step in `pAPT` by having threads keep their own version of the list pointers. Working on the related problem of string similarity joins with edit distance constraints, Jiang et al. [77] provided a parallel version of an earlier algorithm they developed [78], which they named `ParaJoin`. After sorting input strings in parallel, their algorithm builds a number of inverted index structures, designed to match strings of different lengths. Then, all threads share the index structures as each thread processes a subset of the input strings. While they showed improvement over serial methods, `pAPT` and `ParaJoin` were tested using at most 8 threads, on datasets containing mostly short vectors.

The *NN-Descent* algorithm by Dong et al. [36], which we describe in Section 3.2, provides one of the few existing multi-core cosine k-NNG construction solutions that can be used with sparse real-valued vectors. Pinar and Heath [79], and Buluç and coauthors [80,81] proposed general data structures and algorithms for fast computation of sparse matrix vector products which can scale well with an increasing number of cores. While these algorithms may scale better than our tested `pKIdxJoin` baseline, which performs tiled sparse matrix-vector products to solve the k-NNG construction problem, neither of the proposed algorithms takes advantage of any pruning and will thus not be competitive enough against our proposed method.

Chapter 4

Serial ϵ-NNG Construction

In [18], we addressed the problem of constructing the ϵ-NNG where object proximity is measured by *cosine similarity*. We proposed new filtering strategies that successfully prune most candidates that are not neighbors in the constructed graph, resulting in relatively few object pairs having their similarity value computed in full. While previous algorithms do not scale well as the similarity threshold decreases, our new pruning techniques make our method effective at both high and low similarity thresholds.

As discussed in Section 3.1, `AllPairs` is an algorithm for exact all-pairs similarity search introduced by Bayardo et al. [8] and extended by many other filtering-based APSS approaches. By iteratively building a partial inverted index and leveraging several upper bounds on the similarity, `AllPairs` is able to prune away a large number of false positive candidates and achieve significant speedups, especially for datasets with high variance in vector sizes [27]. Our method, `L2AP`, improves over `AllPairs` by obtaining tighter similarity bounds in all stages of the algorithm. We will first detail the filtering framework in `AllPairs` and subsequent extensions, and then present our improvements.

4.1 Filtering Framework

One could solve the APSS problem by finding all nearest neighbors in the dataset for each vector. However, given a sparse dataset, a *query* object d_q may not have features in common with many candidate objects. `AllPairs` avoids computing the similarity of d_q with these objects by using an inverted index, a set of lists, one for

Algorithm 1 The AllPairs algorithm.

```
1: function AllPairs(D, ε)
2:    Process objects in decreasing ‖·‖∞ order
3:    Process features in decreasing frequency order
4:    O ← ∅, I_j ← ∅, for j = 1, ..., m
5:    for each q = 1, ..., n do
6:       O ← O ∪ FindMatchesAP(d_q, I, ε)
7:       b_1 ← 0
8:       for each j = 1, ..., m, s.t. d_{q,j} > 0 do
9:          b_1 ← b_1 + d_{q,j} min(‖f_j‖∞, ‖d_q‖∞)
10:         if b_1 ≥ ε then
11:            I_j ← I_j ∪ {(d_q, d_{q,j})}
12:            d_{q,j} ← 0
13: return O
```

each feature, containing vectors with non-zero values in D for that feature, and their associated feature values. One can then traverse the inverted lists for only the terms in $\mathbf{d}_q$ to find its possible neighbors. Score accumulation (using a map-like data structure to simultaneously keep track of multiple computed scores) using the values stored in the index can be used to compute the similarity value, and the original vector can be discarded [82].

Cosine similarity is invariant to changes in vector lengths. As a result, we can assume that vectors associated with input objects have been normalized to have unit length ($\|\mathbf{d}\|_2 = 1,\ \forall\ d$ in D). The similarity computation then reduces to finding the dot-product between pairs of vectors. Because cosine similarity is commutative, one does not need to compute both $\mathrm{C}(\mathbf{d}_q, \mathbf{d}_c)$ and $\mathrm{C}(\mathbf{d}_c, \mathbf{d}_q)$. To find all objects in the neighborhood of d_q, the index only needs to contain features for previously processed objects. This gave rise to Sarawagi and Kirpal's idea to build the index dynamically [83]. For a given object d_q, one first finds neighbors for d_q using the current version of the index, and then indexes $\mathbf{d}_q$ before moving on to the next object.

`AllPairs` improves these standard similarity search techniques in several ways. It exploits the threshold ϵ and a predefined object processing order to limit the feature values being indexed, the candidate pairs being generated, and for which candidate pairs the exact similarity value should be computed. Algorithms 1 and 2 present the pseudo-code for `AllPairs`. As we continue, we will also detail pruning strategies employed in subsequent extensions `APT` and `MMJoin`.

Algorithm 2 AllPairs FindMatches.

function FINDMATCHESAP($\mathbf{d}_q, \mathcal{I}, \epsilon$)
  $A \leftarrow \varnothing$ ▷ accumulator array
  $M \leftarrow \varnothing$ ▷ set of matches
  $sz_1 \leftarrow \epsilon / \|\mathbf{d}_q\|_\infty$
  $rs_1 \leftarrow \sum_{j=1}^{m} d_{q,j} \|\mathbf{f}_j\|_\infty$
  for each $j = m, \ldots, 1$, s.t. $d_{q,j} > 0$ **do**
    $I_j \leftarrow I_j \setminus \{(d_c, d_{c,j})\},\ \forall\ d_c$ s.t. $\|\mathbf{d}_c\|_0 \leq sz_1$
    for each $(d_c, d_{c,j}) \in I_j$ **do**
      if $A[d_c] > 0$ **or** $rs_1 \geq \epsilon$ **then**
        $A[d_c] \leftarrow A[d_c] + d_{q,j} d_{c,j}$
    $rs_1 \leftarrow rs_1 - d_{q,j} \|\mathbf{f}_j\|_\infty$
  for each d_c s.t. $A[d_c] > 0$ **do**
    if $A[d_c] + \min(\|\mathbf{d}_q\|_0, \|\mathbf{d}_c^{\leq}\|_0) \|\mathbf{d_q}\|_\infty \|\mathbf{d}_c^{\leq}\|_\infty \geq \epsilon$ **then**
      $s \leftarrow A[d_c] + \left\langle \mathbf{d}_q, \mathbf{d}_c^{\leq} \right\rangle$
      if $s \geq \epsilon$ **then**
        $M \leftarrow M \cup \{(d_q, d_c, s)\}$
return M

4.1.1 Prefix and Suffix Filtering

Chaudhuri et al. introduced the *prefix-filtering* principle, which has been used to limit the size of the inverted index. It states informally that, given a global feature processing order, one can stop indexing features in $\mathbf{d}_q$ as soon as they can ensure that $\mathbf{d}_q$ will have at least one feature in the index in common with all its *true neighbors* (those vectors $\mathbf{d}_c$ s.t. $\mathrm{C}(\mathbf{d}_q, \mathbf{d}_c) \geq \epsilon$). Chaudhuri et al. and Lee et al. order their datasets in increasing column frequency order and index features at the beginning of $\mathbf{d}_q$, i.e. its *prefix*. They use the remaining part of the vector, its *suffix*, to estimate and complete similarity computations. While they do not expressly state it, Bayardo et al. also use the *prefix-filtering* principle in their algorithm, `AllPairs`. Yet they choose the opposite order for processing features, index the suffix of each vector, and use the prefix to complete the similarity computation. To avoid confusion, we will refer to prefix filtering, henceforward, as *index filtering*, since its goal is to reduce the index size. Similarly, we will refer to suffix filtering as *residual filtering*, since it operates on the remaining (un-indexed) portion of the vector.

4.1.2 Index Construction

Lines 3 and 7–12 in Algorithm 1 highlight the index size reduction via *index filtering* in `AllPairs`. The algorithm does not start indexing feature values from $\mathbf{d}_q$ until the variable b_1 reaches the similarity threshold ϵ. Once a value is indexed, it is erased form $\mathbf{d}_q$ (line 12). Bayardo et. al [8] show that enough features will be indexed using this method to ensure that any vector $\mathbf{d}_c$ that has the potential to meet the similarity threshold ϵ against $\mathbf{d}_q$ will be identified during the similarity search. While a certain feature processing order is not necessary, processing them in decreasing order of the number of objects a feature is found in (decreasing frequency order, line 3) tends to lead to fewer indexed values.

The variable b_1, which we call the `pscore` (prefix score), captures an upper bound on the similarity score attainable by matching the first features in $\mathbf{d}_q$ against any other vector in the dataset. It is akin to the similarity of $\mathbf{d}_q$ with the maximum possible valued vector in the dataset, which should be computed as $\sum_{j=1}^{m} \|\mathbf{f}_j\|_\infty d_{q,j}$, where $\mathbf{f}_j$ is the vector made up of all the object values for the jth feature. `AllPairs` takes advantage of an imposed object processing order to improve this bound. By processing objects in decreasing order of maximum object values (line 2 of Algorithm 1), one obtains a sharper estimate on a candidate's feature value. The objects we are interested in, which are those that follow d_q in the processing order, are thus guaranteed to have the maximum value for feature j of $\min(\|\mathbf{f}_j\|_\infty, \|\mathbf{d}_q\|_\infty)$.

Awekar and Samatova focus on candidate pruning in `APT`, and make no changes to the index reduction proposed in `AllPairs`. Lee et al., however, achieve better index reduction in `MMJoin` by using the non-negativity of the square of a real number property, $(a-b)^2 \geq 0 \Rightarrow a^2 + b^2 \geq 2ab$. Using this inequality, they derive

$$\begin{aligned} \langle \mathbf{d}_q, \mathbf{d}_c \rangle = \sum\nolimits_l d_{q,l} d_{c,l} &\leq \sum\nolimits_l \frac{{d_{q,l}}^2 + {d_{c,l}}^2}{2} \\ &= \frac{1}{2}\|\mathbf{d}_q\|_2^2 + \frac{1}{2}\|\mathbf{d}_c\|_2^2, \end{aligned} \tag{4.1}$$

that also holds for prefixes or suffixes of vectors at a common feature p, i.e.:

$$\langle \mathbf{d}_q^{\leq p}, \mathbf{d}_c^{\leq p} \rangle \leq \frac{1}{2}\|\mathbf{d}_q^{\leq p}\|_2^2 + \frac{1}{2}\|\mathbf{d}_c^{\leq p}\|_2^2. \tag{4.2}$$

As all vectors are unit length normalized, the dot-product of $\mathbf{d}_q$ with any other vector can then be approximated by $\langle \mathbf{d}_q, \cdot \rangle \leq \frac{1}{2}\|\mathbf{d}_q\|_2^2 + \frac{1}{2}$, which provides another upper bound for the `pscore`. `MMJoin` combines this new bound with the original one in `AllPairs` by using the minimum of the two upper bounds, $\min(b_1, b_2) \geq \epsilon$ in line 10 of Algorithm 1, where $b_2 = \frac{1}{2}\|\mathbf{d}_q^{\leq j}\|_2^2 + \frac{1}{2}$. Lee et al. also use Equation 4.2 during candidate generation and verification, and thus store the value $\frac{1}{2}\|\mathbf{d}_q^{<j}\|_2^2$, in addition to $d_{q,j}$, for each indexed term (line 11 of Algorithm 1 becomes $I_j \leftarrow I_j \cup \{(d_q, d_{q,j}, \frac{1}{2}\|\mathbf{d}_q^{<j}\|_2^2)\}$).

Note that our explanation and notation of Lee et al.'s algorithm has been adjusted to follow the column ordering in `AllPairs`. Their original presentation follows the opposite column ordering. Therefore, they initially pre-compute $b_1 \leftarrow \sum_{j=1}^{m} d_{q,j} \min(\|\mathbf{f}_j\|_\infty, \|\mathbf{d}_q\|_\infty)$ in line 7 of Algorithm 1, and then roll back the computation, indexing until b_1 falls below ϵ.

4.1.3 Candidate Generation

Candidate generation and verification in `AllPairs` are detailed in Algorithm 2. `AllPairs` uses a lower bound (sz_1), which we call `minsize`, to eliminate unpromising indexed objects that have too few values (lines 4 and 7). Bayardo et al. name this process *size filtering*. They show that any candidate vector must have at least $\epsilon/\|\mathbf{d}_q\|_\infty$ non-zero values to possibly achieve ϵ similarity with $\mathbf{d}_q$. Additionally, since objects are processed in decreasing order of their maximum value, the minimum candidate size increases monotonically with each iteration. Those objects that fail this check will then fail it for all future processed objects and can be safely removed from the inverted index (line 7).

`APT` and `MMJoin` both provide stronger bounds for the `minsize` bound. Awekar and Samatova use an upper bound on the dot-product, $\langle \mathbf{d}_q, \mathbf{d}_c \rangle \leq \|\mathbf{d}_q\|_\infty \|\mathbf{d}_c\|_1$ (see Section 2.2), to derive `minsize` as $sz_2 \leq (\epsilon/\|\mathbf{d}_q\|_\infty)^2$. On the other hand, Lee et al. use the upper bound

$$\langle \mathbf{d}_q, \mathbf{d}_c \rangle \leq \min(\|\mathbf{d}_q\|_0, \|\mathbf{d}_c\|_0)\|\mathbf{d}_q\|_\infty \|\mathbf{d}_c\|_\infty$$

to drive it as $sz_3 \leq \epsilon/(\|\mathbf{d}_q\|_\infty \|\mathbf{d}_c\|_\infty)$.

Residual filtering uses an upper bound on the similarity of the un-indexed portion of the vectors, along with the already accumulated dot-product, to prune additional potential candidates. As we accumulate over the features of $\mathbf{d}_q$, there comes a point

when there are not enough features left to allow any new vector without accumulated weight to reach the similarity threshold. `AllPairs` finds this point by maintaining an upper bound `remscore` value (rs_1) on the similarity score that a new vector $\mathbf{d}_c$ (i.e., $A[d_c] = 0$) could achieve with $\mathbf{d}_q$ (lines 5 and 11). Accumulation only starts as long as the `remscore` value is still above the threshold ϵ (line 9). Once accumulation has started for a vector, it becomes a *candidate*.

`APT` uses the same `remscore` bound as in `AllPairs`. `MMJoin` capitalizes on Equation 4.2 in two ways to enhance residual filtering. First, it augments the `remscore` bound in line 9 by checking $\min(rs_1, rs_2) \geq \epsilon$, where $rs_2 = \frac{1}{2}\|\mathbf{d}_q^{\leq j}\|_2^2 + \frac{1}{2}$. Note that accumulation occurs in reverse feature processing order. If no score has yet been accumulated for the object d_c, and $rs_2 < \epsilon$, the similarity between d_q and d_c cannot possibly pass the threshold ϵ, and the potential candidate is skipped. Second, for those candidates that have started accumulating, `MMJoin` pushes a verification step into the candidate generation stage. It keeps checking, after each accumulation change, whether $A[d_c] + \frac{1}{2}\|\mathbf{d}_q^{<j}\|_2^2 + \frac{1}{2}\|\mathbf{d}_c^{<j}\|_2^2$ is below the threshold ϵ. When this estimate falls below ϵ, `MMJoin` stops accumulating d_c (prunes it) and sets $A[d_c] = 0$. Lee et. al. call this process *length filtering*.

4.1.4 Candidate Verification

The similarity $\mathrm{C}(\mathbf{d}_q, \mathbf{d}_c)$ has already been partially computed and stored in the accumulator $A[d_c]$. `AllPairs` then tries to estimate the similarity of the query object with the un-indexed prefix of each candidate $\mathbf{d}_c^{\leq}$ (line 13). This bound, which we call the `dpscore` (dot-product score), allows skipping the full similarity score computation of d_q with the candidate if the estimate is still below ϵ. Otherwise, `AllPairs` computes the remaining similarity between $\mathbf{d}_q$ and the prefix $\mathbf{d}_c^{\leq}$ exactly and adds the pair to the result M as necessary (lines 15-16).

Lee et al. employ the same `dpscore` bound as in `AllPairs`. Leveraging the dot-product upper bound they considered in the `minsize` estimation, Awekar and Samatova propose a new `dpscore` bound, which they prove is a tighter bound than that of Bayardo et al., and is given by:

$$\left\langle \mathbf{d}_q, \mathbf{d}_c \right\rangle \leq A[d_c] + \min(\|\mathbf{d}_q\|_\infty \|\mathbf{d}_c^{\leq}\|_1, \|\mathbf{d}_c^{\leq}\|_\infty \|\mathbf{d}_q\|_1).$$

As an alternate means of pruning, based on their *length filtering* idea, Lee et al. use the prefix similarity estimates they stored in the index and check whether $A[d_c] + \frac{1}{2}\|\mathbf{d}_q^{<j}\|_2^2 + \frac{1}{2}\|\mathbf{d}_c^{<j}\|_2^2$ drops below the threshold while computing the rest of the dot-product, after each accumulation operation. To alleviate excessive checking, they only test this bound at every other feature that the query and candidate objects have in common. The candidate d_c is pruned if the bound falls below ϵ.

4.2 Cosine ϵ-NNG Construction

We now present our algorithm, `L2AP`, which leverages the Cauchy-Schwarz inequality to obtain tighter ℓ_2-norm similarity estimate bounds for both index reduction and candidate generation and verification. In addition, `L2AP` improves on and introduces new residual filtering techniques that help eliminate the majority of candidates before fully computing their similarity value.

4.2.1 ℓ_2-norm Bounds

The majority of the improvement in `L2AP` is due to much tighter bounds obtained by leveraging the Cauchy-Schwarz inequality in partial dot-product estimations. Recall that, $\langle \mathbf{d}_q, \mathbf{d}_c \rangle = \langle \mathbf{d}_q^{\leq}, \mathbf{d}_c \rangle + \langle \mathbf{d}_q^{>}, \mathbf{d}_c \rangle$, where $\mathbf{d}_q^{\leq}$ is the prefix, or un-indexed portion of the vector, and $\mathbf{d}_q^{>}$ is its suffix. By the Cauchy-Schwarz inequality we have that:

$$\langle \mathbf{d}_q^{\leq}, \mathbf{d}_c \rangle \leq \|\mathbf{d}_q^{\leq}\|_2 \|\mathbf{d}_c\|_2. \tag{4.3}$$

Since all vectors are unit length normalized, the prefix dot-product can then be approximated by $\langle \mathbf{d}_q^{\leq}, \mathbf{d}_c \rangle \leq \|\mathbf{d}_q^{\leq}\|_2$. This new bound has profound consequences during indexing and candidate generation. Vectors are accumulated in reverse feature processing order. If $\|\mathbf{d}_q^{\leq}\|_2 < \epsilon$, no terms in $\mathbf{d}_q^{\leq}$ can lead to new candidates that have not yet been identified.

The ℓ_2-norm bound is tighter than the one proposed by Lee et al., $\langle \mathbf{d}_q^{\leq}, \mathbf{d}_c \rangle \leq \frac{1}{2}\|\mathbf{d}_q^{\leq}\|_2^2 + \frac{1}{2}$, since

$$\left(\|\mathbf{d}_q^{\leq}\|_2 - 1\right)^2 \geq 0 \;\Rightarrow\; \frac{1}{2}\|\mathbf{d}_q^{\leq}\|_2^2 + \frac{1}{2} \geq \|\mathbf{d}_q^{\leq}\|_2.$$

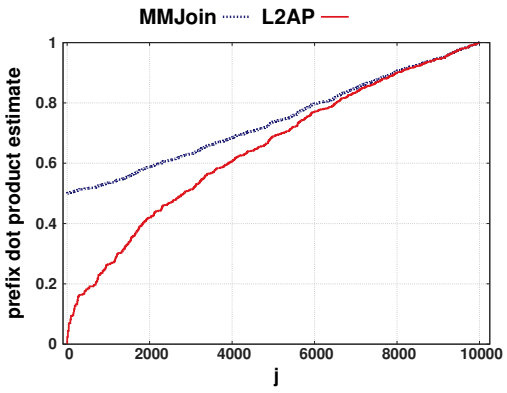

Figure 4.1: Comparison of estimated prefix dot-products $\left\langle \mathbf{d}_q^{\leq j}, \mathbf{d}_c \right\rangle$ for two random 10,000 dimensional vectors in `MMJoin` and `L2AP`.

Their estimate will always exceed 0.5, while ours is closer to the true dot-product. As an example, Figure 4.1 shows the results of computing the prefix dot-product estimates, using our formula and that from `MMJoin`, for two random sparse vectors, at each prefix index j. For $\epsilon = 0.5$, `L2AP` needs 1/3 less features than `MMJoin` to accurately estimate that the prefix similarity of the two vectors drops below ϵ, which is crucial during indexing and candidate generation.

Similarly, an estimate for the dot-product of the exclusive prefixes of $\mathbf{d}_q$ and $\mathbf{d}_c$ at a common term j is given by,

$$\left\langle \mathbf{d}_q^{<}, \mathbf{d}_c^{<} \right\rangle \leq \|\mathbf{d}_q^{<j}\|_2 \|\mathbf{d}_c^{<j}\|_2. \tag{4.4}$$

Candidates can be pruned at a common term j if the sum of their accumulated score and this prefix dot-product estimate falls below the threshold ϵ. Again, this bound is tighter than the similar bound proposed by Lee et al., $\left\langle \mathbf{d}_q^{<}, \mathbf{d}_c^{<} \right\rangle \leq \frac{1}{2}\|\mathbf{d}_q^{<}\|_2^2 + \frac{1}{2}\|\mathbf{d}_c^{<}\|_2^2$, since

$$\left(\|\mathbf{d}_q^{<}\|_2 - \|\mathbf{d}_c^{<}\|_2\right)^2 \geq 0 \;\Rightarrow\; \frac{1}{2}\|\mathbf{d}_q^{<}\|_2^2 + \frac{1}{2}\|\mathbf{d}_c^{<}\|_2^2 \geq \|\mathbf{d}_q^{<}\|_2 \|\mathbf{d}_c^{<}\|_2.$$

4.2.2 Index Construction

Algorithm 3 delineates our proposed method, `L2AP`. We will now highlight the improvements we introduce over the AP framework we discussed in Section 4.1.

Algorithm 3 The L2AP algorithm.

```
1: function L2AP(D, ϵ)
2:     Process objects in decreasing ‖·‖∞ order
3:     Process features in decreasing frequency order
4:     O ← ∅; I_j ← ∅; ĉw ← 0, for j = 1, ..., m
5:     for each i = 1, ..., n do
6:         O ← O ∪ FindMatchesL2AP(d_q, I, ps, ĉw, ϵ)
7:         ĉw_j ← max(d_{q,j}, ĉw_j), ∀ d_{q,j} > 0
8:         b_1 ← 0; b_t ← 0; b_3 ← 0
9:         for each j = 1, ..., m, s.t. d_{q,j} > 0 do
10:            pscore ← min(b_1, b_3)
11:            b_1 ← b_1 + d_{q,j} min(‖f_j‖∞, ‖d_q‖∞)
12:            b_t ← b_t + d²_{q,j}; b_3 ← √b_t
13:            if min(b_1, b_3) ≥ ϵ then
14:                ps[d_q] ← pscore if ps[d_q] = 0
15:                I_j ← I_j ∪ {(d_q, d_{q,j}, ‖d_q^{<j}‖_2)}
16:                d_{q,j} ← 0
17: return O
```

We improve the `pscore` bound in the `AllPairs` framework using our tighter ℓ_2-norm bound. The variable b_3 computes $\|\mathbf{d}_q^{\leq j}\|_2$, the ℓ_2-norm of the prefix of $\mathbf{d}_q$ ending at index j, inclusive. As shown in Section 4.2.1, no new candidates can be identified during accumulation once the prefix norm $\|\mathbf{d}_q^{\leq j}\|$ falls below ϵ. To postpone indexing further, we use the lesser of our new bound, b_3, and the bound proposed by Bayardo et al., to find the minimum number of features we must index. Additionally, we store the exclusive prefix ℓ_2-norm $\|\mathbf{d}_q^{<j}\|$ in the index (line 15), to be used during the candidate generation and verification stages of the algorithm.

The `pscore` bound estimates the similarity of $\mathbf{d}_q^{\leq}$ with any other vector in the dataset. We store the `pscore` value for the query object (lines 10 and 14) and use it during candidate verification as an effective pruning strategy for false positive candidates.

4.2.3 Candidate Generation

Candidate generation and verification in `L2AP` are detailed in Algorithm 4. `L2AP` uses the same `minsize` upper bound as in `MMJoin`[1], $z_3 \leq \epsilon/(\|\mathbf{d}_q\|_\infty \|\mathbf{d}_c\|_\infty)$, which performed

[1]Note that [18] erroneously states that the `MMJoin minsize` bound is superior to the one in `APT`. While both bounds provide limited benefit for different values of ϵ, each can outperform the other for

Algorithm 4 L2AP FindMatches.

1: **function** FINDMATCHESL2AP($\mathbf{d}_q, \mathcal{I}, ps, \hat{cw}, \epsilon$)
2: $A \leftarrow \varnothing$; $M \leftarrow \varnothing$
3: $sz \leftarrow \epsilon / \|\mathbf{d}_q\|_\infty$
4: $rs_3 \leftarrow \|\mathbf{d}_q\|_1 \hat{cw}$; $rs_t \leftarrow 1$; $rs_4 \leftarrow 1$
5: **for each** $j = m, \ldots, 1$, s.t. $x_j > 0$ **do**
6: $I_j \leftarrow I_j \setminus \{(d_c, d_{c,j}, \|\mathbf{d}_c^{<j}\|_2)\}$, $\forall\ d_c$ s.t. $\|\mathbf{d}_c\|_0 \|\mathbf{d}_c\|_\infty \leq sz$
7: **for each** $(d_c, d_{c,j}, \|\mathbf{d}_c^{<j}\|_2) \in I_j$ **do**
8: **if** $A[d_c] > 0$ **or** $\min(rs_3, rs_4) \geq \epsilon$ **then**
9: $A[d_c] \leftarrow A[d_c] + d_{q,j} d_{c,j}$
10: **if** $A[d_c] + \|\mathbf{d}_q^{<j}\|_2 \|\mathbf{d}_c^{<j}\|_2 < \epsilon$ **then**
11: $A[d_c] \leftarrow 0$
12: $rs_3 \leftarrow rs_3 - d_{q,j} \hat{cw}_j$
13: $rs_t \leftarrow rs_t - d_{q,j}^2$; $rs_4 \leftarrow \sqrt{rs_t}$
14: **for each** d_c s.t. $A[d_c] > 0$ **do**
15: **next** d_c **if** $A[d_c] + ps[d_c] < \epsilon$
16: $e1 \leftarrow \min(\|\mathbf{d}_q\|_\infty \|\mathbf{d}_c^{\leq}\|_1, \|\mathbf{d}_c^{\leq}\|_\infty \|\mathbf{d}_q\|_1)$
17: **next** d_c **if** $A[d_c] + e1 < \epsilon$
18: Find greatest p s.t. $d_{c,p}$ in $\mathbf{d}_c^{\leq} \wedge d_{q,p} > 0 \wedge d_{c,p} > 0$
19: $e2 \leftarrow \min(\|\mathbf{d}_q^{\leq p}\|_\infty \|\mathbf{d}_c^{\leq p}\|_1, \|\mathbf{d}_c^{\leq}\|_\infty \|\mathbf{d}_q^{\leq p}\|_1)$
20: **next** d_c **if** $A[d_c] + e2 < \epsilon$
21: **for each** $j < p$ s.t. $d_{c,p}$ in $\mathbf{d}_c^{\leq}$ **do**
22: $A[d_c] \leftarrow A[d_c] + d_{q,j} d_{c,j}$
23: **if** $A[d_c] + \|\mathbf{d}_q^{<j}\|_2 \|\mathbf{d}_c^{<j}\|_2 < \epsilon$ **then**
24: **next** d_c
25: **if** $A[d_c] \geq \epsilon$ **then**
26: $M \leftarrow M \cup \{(d_q, d_c, A[d_c])\}$
27: **return** M

better in our experiments than the respective bound in `APT`, $sz_2 \leq \epsilon^2 / \|\mathbf{d}_q\|_\infty^2$. While we could check both `minsize` bounds, we have found this strategy does not provide additional savings, as the `minsize` bound is not particularly effective as compared to the other bounds we check.

The `remscore` bound enables our algorithm to stop adding new candidates once the estimated dot-product between the prefix of $\mathbf{d}_q$ and all possible candidates falls below ϵ. We improve this bound in two ways. First, note that the similarity of $\mathbf{d}_q$ is computed only against vectors in the inverted index, which come before it in dataset processing order. We use a tighter feature maximum value, $\hat{cw}_j$, in rs_3, an *enhanced* version of Bayardo's proposed `remscore` bound (line 4 of Algorithm 4), which is computed only

different datasets and ϵ options.

over those vectors in the inverted index. Each $\hat{cw}_j$ is updated to the new maximum value after completing the current search (line 7 of Algorithm 3).

Our second improvement involves the ℓ_2-norm bound we discussed in Section 4.2.1. The variable rs_4 uses Equation 4.3 to estimate the dot-product of $\mathbf{d}_q^{\leq j}$, the inclusive prefix of $\mathbf{d}_q$ ending at term j, with any other vector. As long as $\|\mathbf{d}_q^{\leq j}\|$ is not below our threshold ϵ, we can start accumulating a similarity value for a new candidate (line 8 of Algorithm 4).

We use the lesser of rs_3 and rs_4 for our `remscore` bound. While rs_3 can at times be a tighter bound than rs_4, we estimate that most of the time rs_4 will provide a better prefix similarity estimate. At some index p, the two bounds are computed as $rs_3^p = \sum_{j=1}^{p} d_{q,j}\hat{cw}_j$ and $rs_4^p = \sqrt{\sum_{j=1}^{p} d_{q,j}d_{q,j}}$. For most values, $\hat{cw}_j \gg d_{q,j}$, especially given the decreasing maximum value ordering of the vectors, which will likely lead to $rs_3^p > rs_4^p$.

Similar to `MMJoin`, we push a verification step into the candidate generation portion of our algorithm. Based on Equation 4.4, after each accumulation operation, we check whether adding the estimated exclusive prefix similarity, $\langle \mathbf{d}_q^{<}, \mathbf{d}_c^{<} \rangle = \|\mathbf{d}_q^{<j}\|\|\mathbf{d}_c^{<j}\|$, to the accumulated score will be enough to meet the threshold (line 10 of Algorithm 4). If this check fails, we cease accumulating d_c and move to the next candidate.

4.2.4 Candidate Verification

We introduce a new type of candidate pruning, based on the `pscore` bound we computed during indexing, which we call `pscore` *filtering*. At the end of the candidate generation stage, the accumulator $A[d_c]$ contains a partial dot-product, $\langle \mathbf{d}_q, \mathbf{d}_c^{>} \rangle$. Recall that the `pscore` bound estimated the dot-product between the prefix of d_c and any other vector in the dataset, $\langle \mathbf{d}_c^{\leq}, \cdot \rangle$. We stored this estimate at the end of indexing $\mathbf{d}_c$ and use it here for candidate verification, for an estimate of $\langle \mathbf{d}_c^{\leq}, \mathbf{d}_q \rangle$. If the sum of the accumulated score and the estimate falls below ϵ, the candidate is discarded (line 15).

We adopt the `dpscore` bound introduced by Awekar and Samatova, and provide several enhancements, similar in spirit to the *positioning filtering* idea of Xiao et al [6]. We efficiently compute the dot-product of $\mathbf{d}_q$ with candidates by pre-hashing the values in $\mathbf{d}_q$ (we store them in a map data structure). In addition, we choose to also hash prefix maximum and prefix sum values of $\mathbf{d}_q$ at each position j where $d_{q,j} > 0$, which aid in

strengthening the `dpscore` bound. Once the first common feature p is found between $\mathbf{d}_q$ and $\mathbf{d}_c^{\leq}$ (line 18), the following possible `dpscore` variants can be used,

$$\langle \mathbf{d}_q, \mathbf{d}_c \rangle \leq A[d_c] + \min(\|\mathbf{d}_q\|_\infty \|\mathbf{d}_c^{\leq}\|_1, \|\mathbf{d}_c^{\leq}\|_\infty \|\mathbf{d}_q\|_1), \tag{4.5}$$

$$\langle \mathbf{d}_q, \mathbf{d}_c \rangle \leq A[d_c] + \min(\|\mathbf{d}_q^{\leq p}\|_\infty \|\mathbf{d}_c^{\leq p}\|_1, \|\mathbf{d}_c^{\leq}\|_\infty \|\mathbf{d}_q\|_1), \tag{4.6}$$

$$\langle \mathbf{d}_q, \mathbf{d}_c \rangle \leq A[d_c] + \min(\|\mathbf{d}_q\|_\infty \|\mathbf{d}_c^{\leq p}\|_1, \|\mathbf{d}_c^{\leq}\|_\infty \|\mathbf{d}_q^{\leq p}\|_1), \tag{4.7}$$

$$\langle \mathbf{d}_q, \mathbf{d}_c \rangle \leq A[d_c] + \min(\|\mathbf{d}_q^{\leq p}\|_\infty \|\mathbf{d}_c^{\leq p}\|_1, \|\mathbf{d}_c^{\leq}\|_\infty \|\mathbf{d}_q^{\leq p}\|_1). \tag{4.8}$$

Similar enhancements are possible for Bayardo's `L2AP` bound. By hashing prefix maximum and prefix size values in addition to the values of $\mathbf{d}_q$, we can utilize the following bounds once the first common feature p between $\mathbf{d}_q$ and $\mathbf{d}_c^{\leq}$ is found,

$$\langle \mathbf{d}_q, \mathbf{d}_c \rangle \leq A[d_c] + \min(\|\mathbf{d}_q\|_0, \|\mathbf{d}_c^{\leq}\|_0) \|\mathbf{d}_q\|_\infty \|\mathbf{d}_c^{\leq}\|_\infty, \tag{4.9}$$

$$\langle \mathbf{d}_q, \mathbf{d}_c \rangle \leq A[d_c] + \min(\|\mathbf{d}_q\|_0, \|\mathbf{d}_c^{\leq p}\|_0) \|\mathbf{d}_q^{\leq p}\|_\infty \|\mathbf{d}_c^{\leq p}\|_\infty, \tag{4.10}$$

$$\langle \mathbf{d}_q, \mathbf{d}_c \rangle \leq A[d_c] + \min(\|\mathbf{d}_q^{\leq p}\|_0, \|\mathbf{d}_c^{\leq p}\|_0) \|\mathbf{d}_q\|_\infty \|\mathbf{d}_c^{\leq p}\|_\infty, \tag{4.11}$$

$$\langle \mathbf{d}_q, \mathbf{d}_c \rangle \leq A[d_c] + \min(\|\mathbf{d}_q^{\leq p}\|_0, \|\mathbf{d}_c^{\leq p}\|_0) \|\mathbf{d}_q^{\leq p}\|_\infty \|\mathbf{d}_c^{\leq p}\|_\infty. \tag{4.12}$$

Note that Equations 4.5 and 4.9 can be used before finding the first common feature p. One could also try the cheaper bound in Equation 4.5 or Equation 4.9 (line 17), followed by one of the position-based bounds in case of failure (line 20). Equations 4.8 and 4.12 provide the tightest bounds among their respective variants, since $\|\mathbf{d}_q^{\leq p}\|_\infty \leq \|\mathbf{d}_q\|_\infty$, $\|\mathbf{d}_q^{\leq p}\|_1 \leq \|\mathbf{d}_q\|_1$, and $\|\mathbf{d}_q^{\leq p}\|_0 \leq \|\mathbf{d}_q\|_0$. A similar proof as provided by Awekar and Samatova (Section 4.4 in [43]), showing that Equation 4.5 provides a better bound than Equation 4.9, can be constructed to show the superiority of Equation 4.8 for candidate pruning, making it the best of the eight proposed `L2AP` bounds.

If a candidate passes these initial checks, we compute the full dot-product of its remaining prefix with the query vector (lines 21-24). After each accumulation, however, we use our ℓ_2-norm based prefix similarity estimate to further prune unpromising candidates (lines 23-24). Surviving candidates have their final similarity value checked against the threshold ϵ and are added to the result M if they meet it.

4.2.5 Approximate ϵ-NNG Construction

Using Bayesian inference, `BayesLSH` finds the probability that the similarity of two candidates is above the threshold ϵ, conditional on the observed event of LSH hash matches. Additionally, it can estimate the similarity value and the probability that the estimate is within δ of the true similarity. Satuluri and Parthasarathy provide a way to tractably perform inference for the Jaccard and cosine similarity functions applied to both binary and weighted vectors. `BayesLSH-Lite` is a less expensive variant of the algorithm that, after examining a fixed number of hashes h, uses the first probability estimate to prune candidates, with a theoretically guaranteed maximum false negative rate r. It then computes the exact similarity value for un-pruned candidate pairs. This strategy has been shown effective, outperforming both `AllPairs` and `LSH`. The remaining details of `BayesLSH` and `BayesLSH-Lite` are beyond the scope of this thesis and can be found in [27].

We are interested in exact similarity values in our problem, so we combine our algorithm with `BayesLSH-Lite` to form an approximate variant, `L2AP-a`. In Algorithm 4, we replace ℓ_2-norm based candidate verification (lines 21-24) with `BayesLSH-Lite` filtering applied to a pair of vectors $\mathbf{d}_q$ and $\mathbf{d}_c$ (lines 6-14 in Algorithm 2 of [27]). We then complete accumulation and similarity threshold checking for un-pruned candidates. Using `BayesLSH-Lite` at this point allows us to take advantage of most of our pruning strategies before resorting to approximate estimation. While `L2AP-a` may over-prune in this step, it will provide exact similarity values for the neighbors it finds.

4.3 Choice of Pruning Strategies

Our strategy, so far, has been to improve and provide new similarity bounds that can lead to better index reduction, candidate generation, and candidate pruning. Many of the bounds we proposed come with the added cost of more hashing or bound computations. In some cases, this cost may outweigh the benefit of a somewhat smaller candidate set or fewer full dot-products being computed. For example, using Equation 4.6 or 4.7 instead of 4.8 for the `L2AP` bound has the benefit of less hashing, while still being a tighter bound than the one in Equation 4.5. With these thoughts in mind, we built our prototypes, `L2AP` and `L2AP-a`, with the ability to choose, at compile time, the pruning

strategies to employ. This gives us the added benefit of being able to check the effectiveness of individual pruning bounds. Table 4.1 summarizes some of the choices available for each bound in our prototype. The symbols b_x, sz_x, and rs_x refer to the respective `pscore`, `minsize`, and `remscore` bounds described in this thesis. We use $dp_1 - dp_8$ to reference the `L2AP` pruning choices in Equations 4.5 through 4.12. The index construction stage is noted as *i.c.* We note ℓ_2-norm filtering at the candidate generation (*c.g.*) and verification (*c.v.*) stages of the algorithm as $l2cg$ and $l2cv$, respectively. We will use this notation to specify pruning strategies in Section 4.4.

Table 4.1: Pruning strategies in `L2AP`.

Stage	**Bound**	**Bound Choices**
i.c.	`pscore`	b_1, $\min(b_1, b_2)$, $\min(b_1, b_3)$
c.g.	`minsize`	sz_1, sz_3
	`remscore`	rs_1, rs_2, rs_3, rs_4, $\min(rs_1, rs_2), \min(rs_1, rs_4), \min(rs_3, rs_4)$
	ℓ_2-norm	$l2cg$
c.v.	`pscore`	ps
	`dpscore`	$\{dp_1, dp_5\}$+$\{dp_2, dp_3, dp_4, dp_6, dp_7, dp_8\}$, $dp_1 \rightarrow dp_8$
	ℓ_2-norm	$l2cv$

4.4 Experimental Evaluation for Cosine ϵ-NNG Construction

Our cosine ϵ-NNG construction experiment results are organized along two directions. First, we test the effectiveness of the pruning bounds we described in Section 4.2 and compare them against the previously introduced bounds. Then, we test the efficiency of our methods in comparison to several baselines.

4.4.1 Baseline Approaches

We compare `L2AP` and `L2AP-a` against the following baseline approaches.

1. `IdxJoin` is a straight-forward baseline that first builds a full inverted index. Then, without performing any pruning, it uses the index to compute exactly the similarity of each vector with all preceding vectors in the dataset.

2. `AllPairs` is a state-of-the-art approach for solving the APSS problem proposed by Bayardo et al. [8], which we detailed in Section 4.1.

3. `MMJoin` enhances `AllPairs` by adding *length filtering* and a tighter `minsize` bound. These enhancements are also detailed in Section 4.1.

4. `AllPairs` +`BayesLSH-Lite` and `LSH` +`BayesLSH-Lite` are state-of-the-art approximate methods proposed by Satuluri and Parthasarathy [27], variants of `BayesLSH` that take as input the candidate set generated by `AllPairs` and `LSH`, respectively. They compute the similarity values exactly but may not return all nearest neighbors. They have been shown to significantly outperform `LSH`, which we do not include in the comparison.

4.4.2 Pruning Effectiveness

We evaluated the effectiveness of ℓ_2-norm filtering in `L2AP`, comparing against previously proposed filtering strategies. First, we provide a summary of our findings. Overall, the newly proposed indexing and pruning bounds were more effective than previous ones. The `pscore` bound used during indexing resulted in much smaller indexes, with the highest reduction in the number of indexed values at $\epsilon = 0.5$. For some datasets, `L2AP` indexed less than 50% of the non-zero values that were indexed by previous methods. Similarly, the new `remscore` bounds lead to fewer objects being considered as candidates, in some cases `L2AP` accumulating similarity scores for less than 30% of the candidate objects considered by other methods. We found that both the reductions in the number of indexed features and candidates were not as pronounced for network datasets.

Our `pscore` candidate filtering strategy as well as the ℓ_2-norm based filtering bounds checked in the candidate generation and verification stages of `L2AP` proved to be effective strategies for eliminating false positive candidates, leading to the majority of candidates being pruned before computing their full dot-product. On the other hand, we found `minsize` and `dpscore` pruning strategies less effective. The `minsize` bounds had no effect for the text based datasets we experimented with. While they pruned some candidates in network datasets, we found that the same candidates were generally pruned by ℓ_2-norm bounds even when `minsize` pruning was not used. Similarly, even though

`dpscore` bounds were effective at pruning some candidates, we did not observe great improvements in efficiency as a result of that pruning.

Effectiveness of the new `pscore` bound for indexing

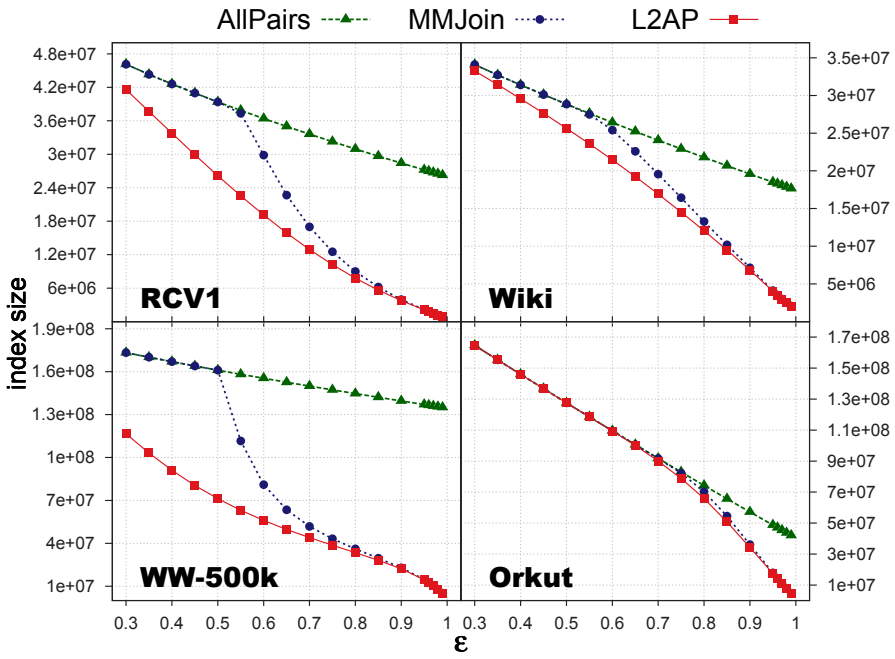

Figure 4.2: Index size reduction in `L2AP` vs. previous methods.

A smaller `pscore` bound at each threshold ϵ leads to Algorithms 1 and 3 starting the indexing process later, thus smaller inverted indexes. We proposed that our ℓ_2-norm based prefix similarity estimate is more effective at lowering this bound than previous strategies. Figure 4.2 shows the index sizes achieved using the `pscore` bound in `L2AP`, `MMJoin`, and the original bound in `AllPairs`. As expected, the `pscore` bound in `L2AP` produces significantly smaller indexes than previous bounds. While the bound in `MMJoin` achieves similar index sizes at high values for ϵ, it degrades to the performance of the `AllPairs` bound as $\epsilon \to 0.5$. Orkut is the only dataset for which our ℓ_2-norm bound is unable to reduce the index size further than the `pscore` bound in `AllPairs`, for $\epsilon < 0.7$.

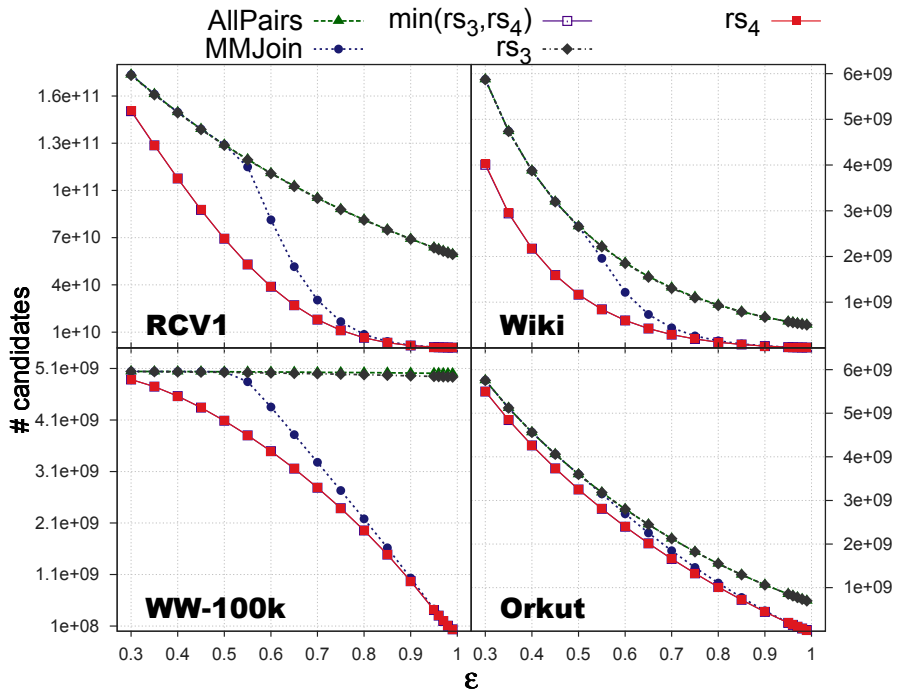

Figure 4.3: Candidate pool sizes when using different `remscore` bounds.

Effectiveness of the new `remscore` bound for candidate generation

In Section 4.2.3, we estimated that our proposed `remscore` bound, rs_4, is tighter than our *enhanced* version of the bound proposed by Bayardo et al., rs_3. To verify this intuition, we counted, during the algorithm execution, how many times rs_3 vs. rs_4 was the minimum value in $\min(rs_3, rs_4)$ (line 8 of Algorithm 4). For this test we used parameters that minimized candidate generation $(\min(b_1, b_3), sz_3)$, and only counted when a new candidate was being generated, i.e. when $\min(rs_3, rs_4) \geq \epsilon$ and $A[d_c] = 0$. The results, which are detailed in Table 4.2, confirmed our estimation. Our new bound, rs_4, was the minimum, averaged over all similarity values, over 97.6% of the times that the `remscore` bound was checked for a new candidate. This suggests that `L2AP` can be effective, and possibly more efficient, using only the ℓ_2-norm part of the `remscore` bound, i.e. $rs_4 \geq \epsilon$ instead of $\min(rs_3, rs_4) \geq \epsilon$ in line 8 of Algorithm 4.

In another test, we compared the effectiveness of the new `remscore` bounds, rs_3, rs_4 and $\min(rs_3, rs_4)$, against previous bounds rs_1 (`AllPairs`) and $\min(rs_1, rs_2)$ (`MMJoin`),

Table 4.2: Performance comparison of the rs_3 and rs_4 bounds.

Dataset	%	**Dataset**	%
RCV1	99.82	Wiki	99.53
WW-500k	99.47	Twitter	98.75
WW-100k	98.05	Orkut	97.61

by counting the number of candidates being generated when using the same indexing strategy as in `AllPairs` (b_1). We did not use any additional pruning or index reduction in this test. Figure 4.3 shows the candidate pool sizes achieved when using the different `remscore` bounds. The enhanced version of Bayardo's bound, rs_3 (almost covering the `AllPairs` line in the figure), is unable to reduce the number of candidates much more than the `AllPairs` bound. On the other hand, rs_4 significantly outperforms both rs_3 and the bound in `MMJoin`, resulting in significant reductions in the candidate pool size. The minimum of the two bounds, $\min(rs_3, rs_4)$, is covered by rs_4 in the figure.

Effectiveness of the new ℓ_2-norm filtering

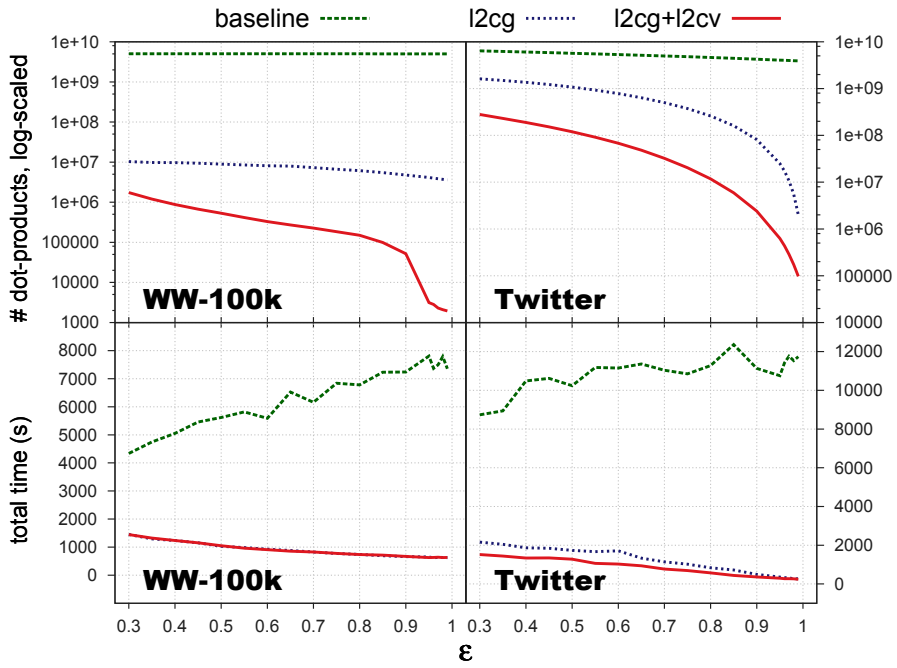

Figure 4.4: Number of dot-products and total time with and w/o ℓ_2-norm filtering.

The ℓ_2-norm based similarity estimation in `L2AP` is the most effective of our pruning

strategies. We have already shown, in Section 4.4.2, that it greatly reduces the size of the inverted index being constructed. We now evaluate the effectiveness of ℓ_2-norm based pruning in the candidate generation and candidate verification stages of our algorithm. For this test, we did not use the ℓ_2-norm during index construction, leveraging only the `AllPairs` bound (b_1) for this step. We tested a *baseline* with no ℓ_2-norm filtering, then add it in the candidate generation stage ($l2cg$), and in the candidate verification stage ($l2cv$) of the algorithm. We record, under each scenario, the number of full dot-products (number of candidates that were not pruned) and total execution time.

As can be seen in Figure 4.4, ℓ_2-norm filtering in `L2AP` is able to drastically reduce the number of dot-products being computed, at times by several orders of magnitude. We find that the majority of the pruning happens in the candidate generation step, and most of the cost associated with this bound is in the initial computation of the prefix magnitude, $\|d^{\leq}\|$, which is stored in the index or hashed. Thus, we find little difference in execution times when enabling ℓ_2-norm filtering in the c.v. stage in addition to the c.g. stage.

Effectiveness of the new `pscore` bound for candidate pruning

We compared our algorithm's execution with and without `pscore` filtering, measuring the number of unpruned dot-products and total execution time, under two experimental scenarios. In the first, we used `AllPairs` bounds in the index reduction and candidate generation stages (b_1, rs_1), allowing `pscore` filtering to be most productive. The `pscore` in this test is based primarily on the dot-product estimate with the maximum possible vector in the dataset, and does not take advantage of the ℓ_2-norm based prefix similarity estimate. We note this baseline without `pscore` filtering as *base1* in Figure 4.5, and the results of this experiment with `pscore` filtering as *pscore1*. In a second experiment, we enabled the most pruning possible in the i.c. and c.g. stages of the algorithm ($\min(b_1, b_3)$, $\min(rs_3, rs_4)$, and $l2cg$), and no other pruning during the c.v. stage. The `pscore` here takes advantage of the ℓ_2-norm based prefix similarity estimate computed during indexing. We note this baseline without `pscore` filtering as *base2*, and the result with `pscore` filtering as *pscore2*.

As shown in Figure 4.5, `pscore` filtering is quite effective at reducing the number of full dot-products, which results in significantly smaller execution times (up to 38%

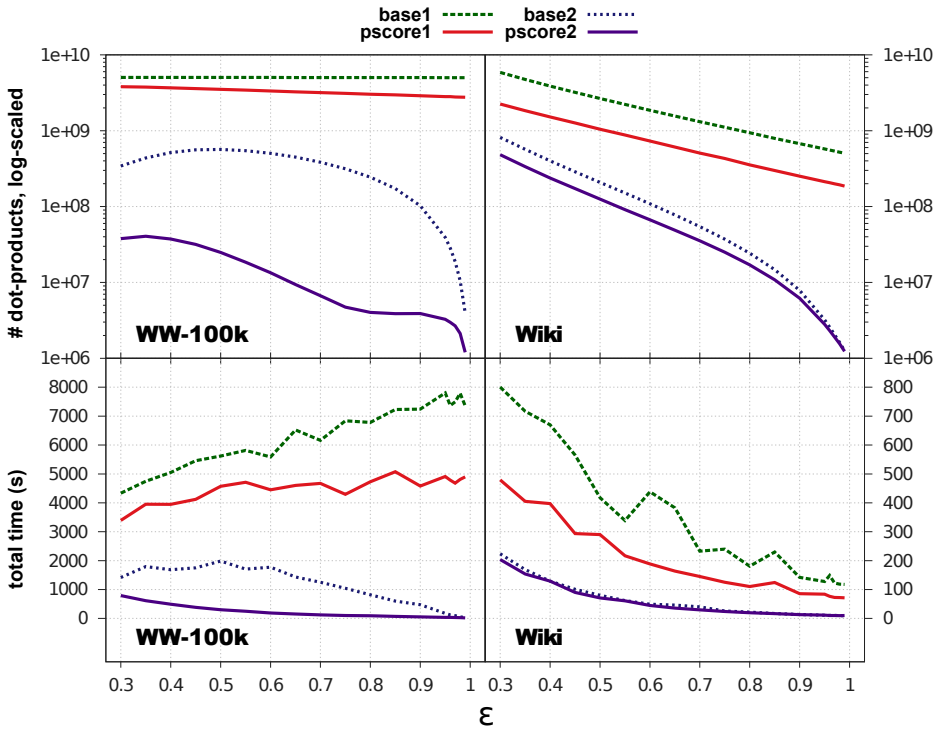

Figure 4.5: Number of dot-products and total time with and w/o `pscore` filtering.

smaller). While its effectiveness is reduced when previous pruning occurs, as expected, `pscore` filtering still reduces execution time by considerable amounts for text datasets.

Effectiveness of the new `L2AP` bounds for positional filtering

Figure 4.6 shows the number of full dot-products and total execution time when pruning using each of the *dp* bounds we proposed. For this test, we employed maximal index reduction and candidate generation pruning ($\min(b_1, b_3)$, $\min(rs_3, rs_4)$, and $l2cg$), and only *dp* pruning during candidate verification. We also included a baseline in which no *dp* pruning was used (*no dp*).

As predicted, dp_4 is able to achieve the best reduction in the number of full dot-products. However, it requires the most hashing and can sometimes lead to longer execution times than other *dp* bounds. Overall, the amount of pruning achieved using the various *dp* bounds only leads to modest reductions in the execution time.

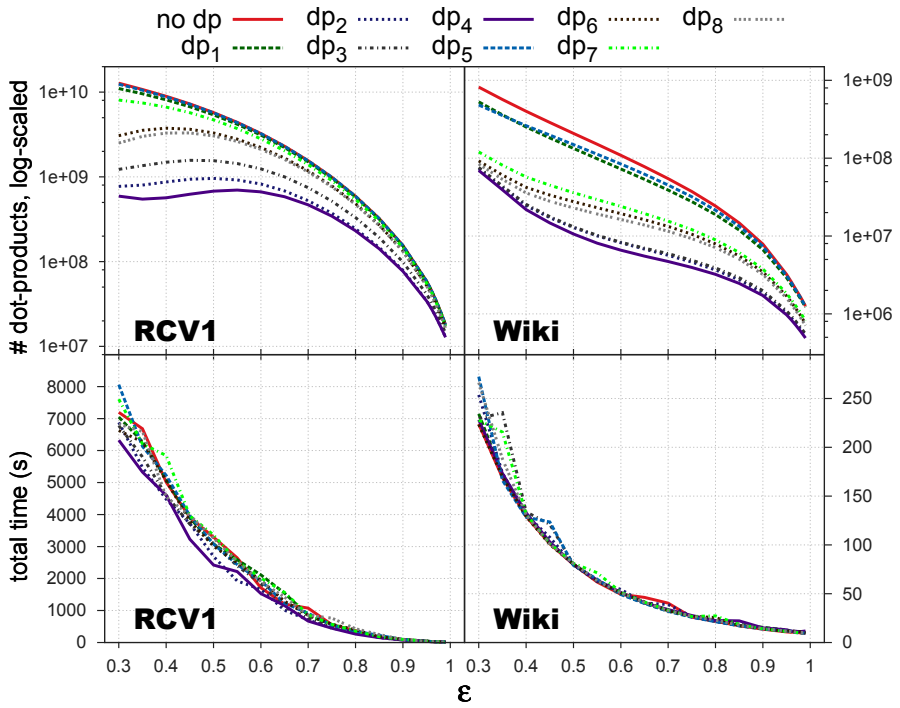

Figure 4.6: Number of dot-products and total time when using different `L2AP` bounds.

We also tested combinations of *dp* bounds, as noted in Table 4.1. Overall, we have found the best *dp* pruning strategy to be $dp_5 + dp_6$ for most datasets. While this strategy is not able to forego as many dot-product computations as dp_4, it does not require computing and storing vector prefix sums, a source of delay in $dp_1 - dp_4$. When testing *dp* pruning in concert with other filtering strategies at the candidate verification stage, we found that, for high similarity values, *dp* pruning is overshadowed by ℓ_2-norm and `pscore` pruning, and becomes ineffective.

A word on the `minsize` bound

Similar to Bayardo et al., we implement inverted lists as arrays and lazily remove vectors pruned by the `minsize` bound, only from the beginning of the lists. Using this strategy, we found that size filtering provided little additional pruning over the other strategies, and in most cases slowed down the overall computation.

4.4.3 Execution Efficiency

In this section, we compare the total execution time of L2AP in relation to exact and approximate baselines. Figures are best viewed in color.

Comparison with exact baselines

In the previous section, we noted the effectiveness of the individual L2AP pruning strategies proposed in this chapter. Each pruning strategy comes with additional bounds computation and checking costs, and leads to efficient similarity search only when it is highly effective at reducing the index size, pruning candidates, or stopping accumulation early for false positive candidates. Combining strategies is not always straight-forward, as pruning in one stage of the algorithm can affect the effectiveness of bounds in later stages. We found the most efficient combination of pruning strategies across datasets and similarity thresholds to be ℓ_2-norm enhanced index construction ($\min(b_1, b_3)$), ℓ_2-norm based candidate generation (rs_4, $l2cg$), and ℓ_2-norm and L2AP filtering in the candidate verification stage (ps, dp_5, dp_6, $l2cv$). We use this pruning strategy across all datasets and similarity thresholds as representative of our algorithm, L2AP.

Figure 4.7 shows the total execution times for L2AP and the other exact baselines, IdxJoin, AllPairs, and MMJoin. We also include results for L2AP*, in which we choose the best performing pruning strategy for each dataset and similarity threshold combination. L2AP* will then always perform as well as or better than L2AP. However, L2AP performs almost as well as it could, given optimal pruning choices. For text datasets, the two schemes have nearly identical timings, the line for L2AP in Figure 4.7 almost completely hiding the one for L2AP*, and their differences are rather small for the other datasets.

L2AP is able to outperform exact baselines in most cases and achieves significant speedups, up to 1600x against AllPairs, and 2x-13x in general over the best exact baseline. Its best performance is at high similarity thresholds, showing its usefulness in tasks such as near-duplicate object detection. In particular, L2AP was able to find pages with nearly the same link profiles among 1.8M English Wikipedia pages in 10 seconds. The most drastic performance difference is between L2AP and AllPairs or IdxJoin at $\epsilon = 0.99$. L2AP's much smaller index and effective candidate pruning strategies allow

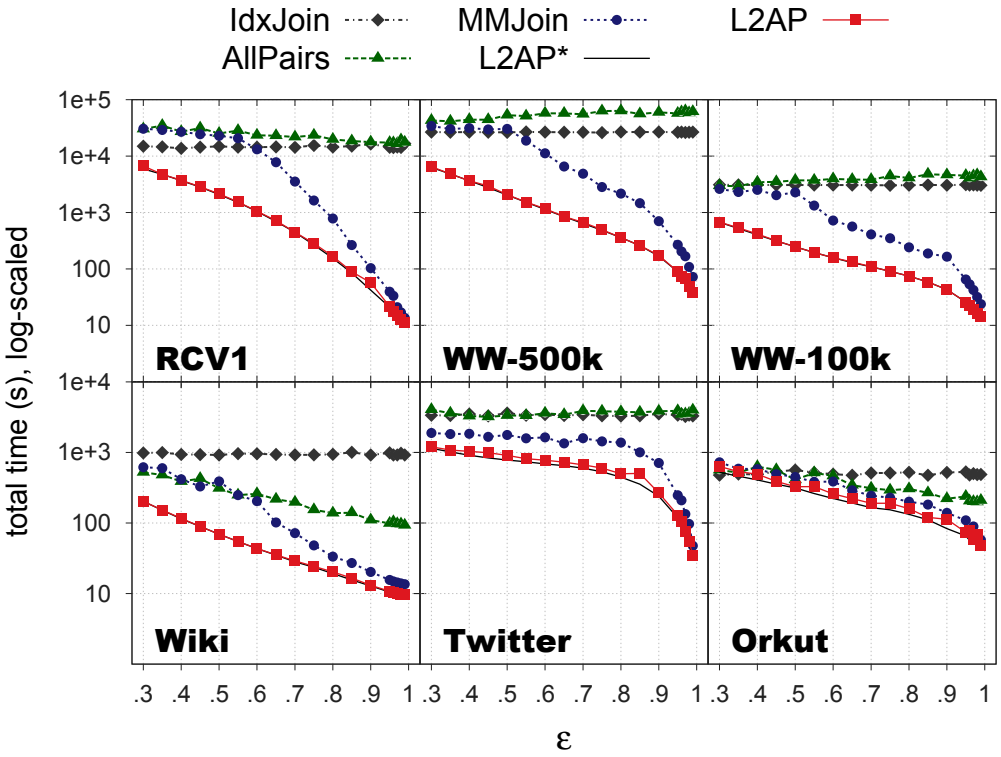

Figure 4.7: Total execution times for exact algorithms.

it to finish the similarity search in a few seconds, while `AllPairs` and `IdxJoin` spend hours to accomplish the same task. An interesting observation is that our straightforward `IdxJoin` baseline, which does no pruning and fully computes vector similarities, outperforms `AllPairs` in several datasets. This shows that excessive bounds checking which does not lead to enough pruning can be detrimental in similarity search.

`MMJoin` uses similar index reduction and pruning strategies as `L2AP`, and is able to achieve comparable performance at high similarity thresholds. `L2AP`'s ℓ_2-*norm filtering* is shown more effective than `MMJoin`'s *length filtering*, however, especially at low similarity thresholds. While `MMJoin` degrades to the same efficiency as `AllPairs` at $\epsilon = 0.5$, `L2AP` is able to finish the task an order of magnitude faster for text datasets.

Link datasets present different challenges, often having much smaller vector and inverted list sizes than text datasets. This limits the effectiveness of the type of pruning that filtering APSS methods utilize. The smaller dimensionality and varied term usage within documents lead to longer inverted lists and better pruning potential in

text datasets. While the speedup is not as dramatic as for text datasets, the pruning strategies in L2AP are effective for link datasets also, achieving up to 4.7x speedup. As Bayardo et al. have also noted [8], Orkut has an artificial 1000 friend limit that prevents highly frequent features, leading to the least possibility of improvement for L2AP over prefix-filtering baselines.

Comparison with approximate baselines

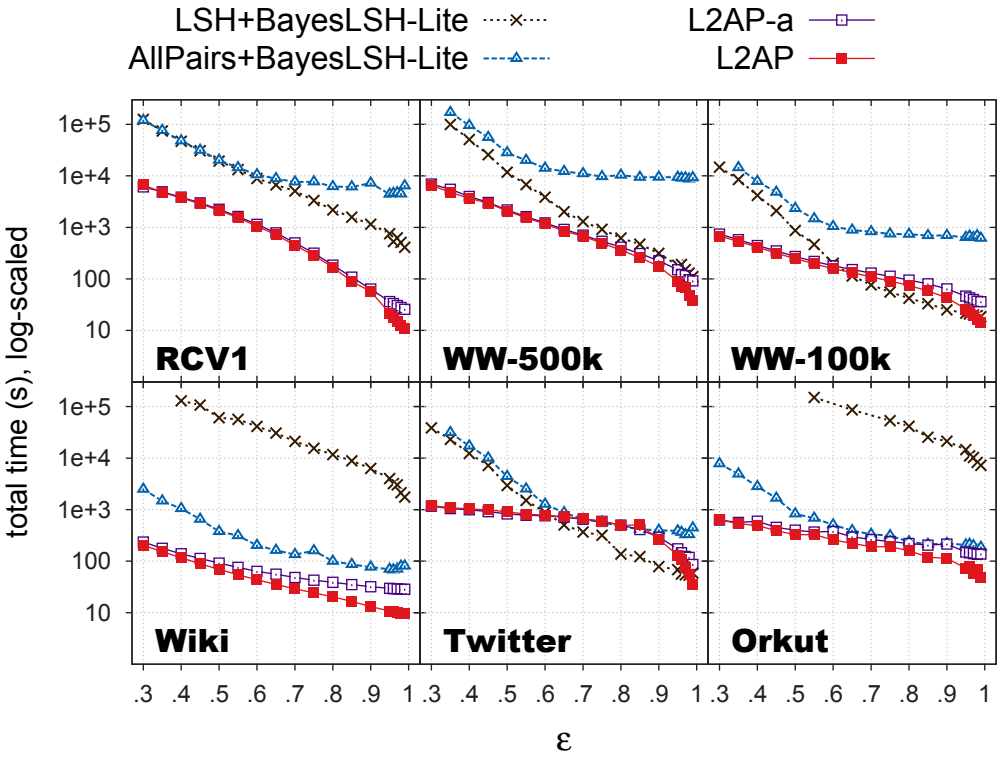

Figure 4.8: Total execution times for approximate algorithms.

Figure 4.8 shows the total execution times for L2AP and L2AP-a as compared with the other approximate baselines. First of all, it is interesting to note that, while their execution times are often close, L2AP outperforms L2AP-a in most cases. L2AP is able to prune most candidates before the approximate BayesLSH-Lite candidate pruning step in L2AP-a. The remaining pruning is not enough to outweigh the cost associated with LSH hashing or Bayesian inference in BayesLSH-Lite.

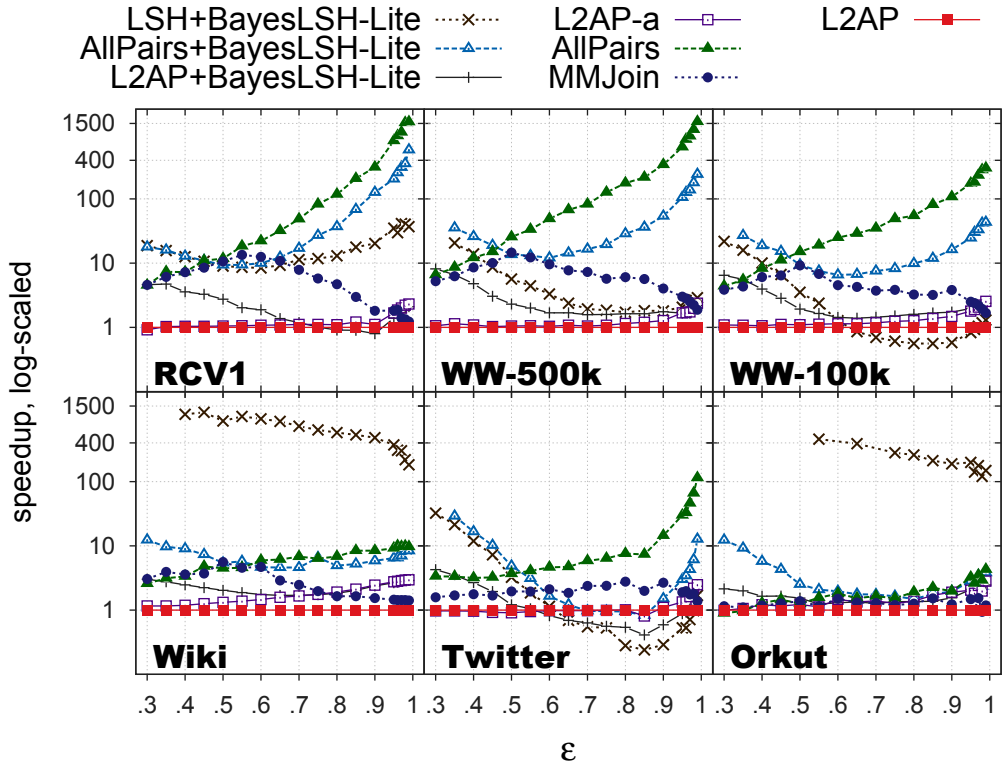

Figure 4.9: `L2AP` speedup over competing methods.

Figure 4.9 gives a different view into the total time comparison, showing speedups obtained by `L2AP` against both exact and approximate baselines. We executed `L2AP-a` with the same pruning parameters we used for `L2AP`, other than $l2cv$, which is replaced by `BayesLSH-Lite` pruning. In addition, we tested a version in which `L2AP` was used only for candidate generation, and `BayesLSH-Lite` was used for candidate verification and pruning, similar to `LSH+BayesLSH-Lite` and `AllPairs+BayesLSH-Lite`. We denote this version in the Figure as `L2AP+BayesLSH-Lite`.

`L2AP` generally outperforms approximate baselines, especially at low similarity thresholds. `LSH+BayesLSH-Lite` outperforms `L2AP` only for the WW-100k and Twitter datasets, and only at similarity values above 0.6. For other datasets, such as Wiki and Orkut, `LSH+BayesLSH-Lite` was not able to finish APSS at low similarities in the time allotted (48 hours). `LSH` degrades quickly for high dimensional datasets and as ϵ decreases, producing large candidate pools that cannot be pruned fast enough even by `BayesLSH-Lite`. In contrast, `L2AP` performs well for all datasets and for both high and low similarity

thresholds, and returns all similar enough object pairs after the search.

4.5 Tanimoto ϵ-NNG Construction

We now describe our algorithm that can be used to exactly compute pairwise Tanimoto similarities with a minimum value of ϵ. First, we describe how a bound on the length of indexed vectors can be efficiently integrated into an inverted index APSS approach. We then show how bounds on the cosine similarity of non-negative real-valued vectors can be used to achieve additional pruning. Finally, we introduce new theoretic bounds on the Tanimoto similarity that rely on partially computed dot-products of vectors, and we show how our method can efficiently use these bounds to eliminate object comparisons.

4.5.1 A Basic Indexing Approach

One approach to find neighbors for a given query object that has been reported to work well in the similarity search literature [8, 15, 18, 43, 49, 53, 54] has been to use an inverted index, which makes it possible to avoid computing similarities between the query and objects that do not have any non-zero features in common with it. As noted in Section 4.1, a map-based data structure, called an *accumulator* [16], can be used to compute the dot-product of the query with all objects encountered while iterating through the inverted lists for non-zero features in the query. We call an object that has a non-zero accumulated dot-product a *candidate*. Using precomputed lengths for the object vectors, the dot-products of all candidates can be transformed into Tanimoto similarities according to Equation 2.22, and those coefficients at or above ϵ can be stored in the output.

One inefficiency with this approach is that it does not take advantage of the commutativity property of the Tanimoto similarity, computing $\text{sim}(d_q, d_c)$ both when accumulating similarities for d_q and for d_c. To address this issue, Bayardo et. al [8] have suggested building the index dynamically, adding the query vector to the index only after finding its neighbors. This ensures that the query is only compared against previously processed objects in a given processing order. While we followed the same approach in designing `L2AP` (see Section 4.2), we now suggest a different approach that

is in general equally efficient, due to the configuration of memory hierarchy in processors today. Given an object processing order, we first re-label each document to match the processing order, then build the inverted index fully, adding objects to the index in the processing order. The result will be inverted lists sorted in non-decreasing order of document labels. Then, when iterating through each inverted list, we can stop as soon as the encountered document label is greater or equal to that of the query. Since the document label will have already been read from memory to perform the accumulation operation and will be resident in the processor cache, the additional check against the value of the query label will be very fast, and will be hidden by the latency associated with loading the next cache line from memory.

Kryszkiewicz [20] has shown that some of the candidate vectors whose lengths are either too small or too large compared to that of the query cannot be its neighbors and can thus be ignored. A candidate d_c cannot be a neighbor of a query object d_q if its length $\|\mathbf{d}_c\|$ falls outside the range $[(1/\alpha)\|\mathbf{d}_q\|,\ \alpha\|\mathbf{d}_q\|]$, where $\|\mathbf{d}_q\|$ is the length of the query vector and

$$\alpha = \frac{1}{2}\left(\left(1+\frac{1}{\epsilon}\right)+\sqrt{\left(1+\frac{1}{\epsilon}\right)^2-4}\right). \tag{4.13}$$

In Section 4.5.3, we show this bound is actually the limit of a new class of Tanimoto similarity bounds we introduce in this thesis. Here, we will show how candidate length pruning can be efficiently integrated into our basic indexing approach.

A candidate object will be encountered as many times in the index as it has non-zero features in common with the query. To avoid checking its length against that of the query each time, we could use a data structure such as a map or bit vector to mark when a candidate has been checked. While checking this data structure may be less computationally demanding than a multiplication and a comparison, it can actually be slower if the number of candidates is high and the data structure does not fit in the processor cache. A better alternative would be to process objects in non-decreasing vector length order. By re-labeling objects as discussed earlier, objects whose lengths are too small will be potentially found at the beginning of the inverted lists, while objects whose lengths are too big will be automatically ignored, as they will come after the query object in the processing order. Note also that, for an object d_c following d_q

in the processing order,

$$\frac{1}{\alpha}\|\mathbf{d}_c\| \geq \frac{1}{\alpha}\|\mathbf{d}_q\|,$$

since $\|\mathbf{d}_c\| \geq \|\mathbf{d}_q\|$ and both vector lengths and α are non-negative real values. As such, the label of the maximum candidate that can be ignored will be non-decreasing. Our approach thus uses a list of starting points, one for each inverted list and updates the starting point of a list each time a new candidate whose length is too small is found in it.

Algorithm 5 TAPNN inverted index approach.

```
1: function TAPNN-1(D, ϵ)
2:     A ← ∅                                                    ▷ accumulator
3:     S ← ∅                                                    ▷ list starts
4:     N ← ∅                                                    ▷ set of neighbors
5:     Compute and store vector lengths for all objects
6:     Permute objects in non-decreasing vector length order
7:     for each q = 1, ..., |D| s.t. ||d_c|| ≤ ||d_q|| ∀c ≤ q do
8:         for each j = 1, ..., m s.t. d_{q,j} > 0 do           ▷ Indexing
9:             I_j ← I_j ∪ {(d_q, d_{q,j})}
10:    for each q = 1, ..., |D| s.t. ||d_c|| ≤ ||d_q|| ∀c ≤ q do
11:        Find label d_max of last object that can be ignored
12:        for each j = 1, ..., m s.t. d_{q,j} > 0 do
13:            for each k = S[j], ..., |I_j| do
14:                (d_c, d_{c,j}) ← I_j[k]
15:                if d_c ≤ d_max then
16:                    S[j] ← S[j] + 1
17:                else if d_c ≥ d_q then
18:                    break
19:                else                                         ▷ Accumulation
20:                    A[d_c] ← A[d_c] + d_{q,j} d_{c,j}
21:        for each d_c s.t. A[d_c] > 0 do                      ▷ Verification
22:            Scale dot-product in A[d_c] according to Equation 2.22
23:            if A[d_c] ≥ ϵ then
24:                N ← N ∪ (d_q, d_c, A[d_c])
25: return N
```

Algorithm 5 provides a pseudo-code sketch for our basic inverted index based approach. The method first permutes objects in non-decreasing vector length order and indexes them. Then, for each query object in the processing order, the maximum object d_{max} satisfying $(1/\alpha)\|\mathbf{d}_{max}\| < \|\mathbf{d}_q\|$ is identified. When iterating through the jth inverted list, `TAPNN` avoids objects in the list whose lengths have already been determined

too small by starting the iteration at index $S[j]$, which is incremented as more objects are found with small lengths. At the end of the accumulation stage, the accumulator contains full dot-products between the query and all objects that could be its neighbors. For each such object, the algorithm computes the Tanimoto similarity using the dot-product stored in the accumulator, and adds the object to the result set if its similarity meets the threshold.

4.5.2 Incorporating Cosine Similarity Bounds

As we have discussed at length in Section 4.2, a number of recent methods have been devised that use similarity bounds to efficiently solve the cosine similarity APSS problem. Moreover, Lee et al. [15] have shown that, for non-negative vectors and the same threshold ϵ, the set of Tanimoto neighbors of an object is actually a subset of its set of cosine neighbors. This can be seen from the formulas of the two similarity functions.

$$\mathrm{T}(d_i, d_j) = \frac{\langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2 - \langle \mathbf{d}_i, \mathbf{d}_j \rangle}$$
$$\mathrm{C}(d_i, d_j) = \frac{\langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\| \|\mathbf{d}_j\|}$$

Given a common numerator, it remains to find a relationship between the denominators in the two functions. Since, for any real valued vector lengths, $(\|\mathbf{d}_i\| - \|\mathbf{d}_j\|)^2 \geq 0$, it follows that,

$$\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2 - 2\|\mathbf{d}_i\|\|\mathbf{d}_j\| \geq 0,$$
$$\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2 - \langle \mathbf{d}_i, \mathbf{d}_j \rangle \geq \|\mathbf{d}_i\|\|\mathbf{d}_j\|,$$

where the last equation follows by applying the Cauchy-Schwarz inequality, which states that $\langle \mathbf{d}_i, \mathbf{d}_j \rangle \leq \|\mathbf{d}_i\|\|\mathbf{d}_j\|$. As a result, the following relationships can be observed between the cosine and Tanimoto similarities of two vectors,

$$\mathrm{T}(d_i, d_j) \leq \mathrm{C}(d_i, d_j),$$
$$\mathrm{T}(d_i, d_j) \geq \epsilon \Rightarrow \mathrm{C}(d_i, d_j) \geq \epsilon,$$
$$\mathrm{C}(d_i, d_j) < \epsilon \Rightarrow \mathrm{T}(d_i, d_j) < \epsilon.$$

One can then solve the Tanimoto APSS problem by first solving the cosine APSS problem and then filtering out those cosine neighbors that are not also Tanimoto neighbors. Given the computed cosine similarity of two vectors and stored vector lengths, the Tanimoto similarity can be derived as follows.

$$\mathrm{T}(d_i, d_j) = \frac{\frac{\langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\|\|\mathbf{d}_j\|}}{\frac{\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2 - \langle \mathbf{d}_i, \mathbf{d}_j \rangle}{\|\mathbf{d}_i\|\|\mathbf{d}_j\|}} = \frac{\mathrm{C}(d_i, d_j)}{\frac{\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2}{\|\mathbf{d}_i\|\|\mathbf{d}_j\|} - \mathrm{C}(d_i, d_j)} \tag{4.14}$$

Note that,

$$(\|\mathbf{d}_i\| + \|\mathbf{d}_j\|)^2 \geq 0 \Rightarrow \frac{\|\mathbf{d}_i\|^2 + \|\mathbf{d}_j\|^2}{\|\mathbf{d}_i\|\|\mathbf{d}_j\|} \geq 2,$$

which provides a higher pruning threshold when searching for cosine neighbors given a Tanimoto similarity threshold ϵ,

$$\mathrm{T}(d_i, d_j) \geq \epsilon \Rightarrow \frac{\mathrm{C}(d_i, d_j)}{2 - \mathrm{C}(d_i, d_j)} \geq \epsilon \Rightarrow \mathrm{C}(d_i, d_j) \geq \frac{2\epsilon}{1 + \epsilon} = t \tag{4.15}$$

Recall that, unlike the Tanimoto similarity, cosine similarity is length invariant. Vectors can thus be normalized as a pre-processing step, which reduces cosine similarity to the dot-product of the normalized vectors. This step, in fact, reduces the number of floating point operations needed to solve the problem, and is standard in cosine APSS methods. Note that the method outlined in Algorithm 5 can also be applied to normalized vectors, adding only a normalization step before indexing and replacing the scaling factor in line 22, using Equation 4.14 instead of Equation 2.22.

In Section 4.2, we described a number of cosine similarity bounds based on the ℓ^2-norm of prefix or suffix vectors that have been found to be more effective than previous known bounds for solving the cosine APSS problem. It may be beneficial to incorporate this type of filtering in our Tanimoto APSS method. However, some of the bounds we used in `L2AP` rely on an object processing order different than the one we proposed for `TAPNN`. We therefore use only ℓ^2-norm based bounds that are processing order independent. This allows our method to still take advantage of the vector length based filtering described in Section 4.5.1. In the remainder of this section, we will describe the ℓ^2-norm based filtering in `TAPNN`.

As shown in Section 4.2.1, the prefix dot-product can be upper-bounded by the length of the prefix vector,

$$\left\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \right\rangle \leq \|\hat{\mathbf{d}}_q^{\leq p}\|. \tag{4.16}$$

Another bound on the prefix dot-product can be obtained by considering the maximum values for each feature among all normalized object vectors. Let $\mathbf{f}_j$ denote the vector of all feature values for the jth feature within the normalized vectors, and $\mathbf{mx}$ the vector of maximum such feature values for each dimension, defined as,

$$\mathbf{f}_j = (\hat{d}_{1,j}, \hat{d}_{2,j}, \ldots, \hat{d}_{n,j}),$$
$$\mathbf{mx} = (\|\mathbf{f}_1\|_\infty, \|\mathbf{f}_2\|_\infty, \ldots, \|\mathbf{f}_m\|_\infty).$$

Then,

$$\left\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \right\rangle = \sum_{l=1}^{m} d_{q,l} d_{c,l} \leq \sum_{l=1}^{m} d_{q,l} mx_l = \langle \hat{\mathbf{d}}_q^{\leq p}, \mathbf{mx} \rangle. \tag{4.17}$$

Combining the bounds in Equation 4.16 and Equation 4.17, we obtain a bound on the prefix similarity of a vector with any other object in D, which we denote by $ps_q^{\leq p}$,

$$\left\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \right\rangle \leq ps^{\leq p} = \min(\|\hat{\mathbf{d}}_q^{\leq p}\|,\ \langle \hat{\mathbf{d}}_q^{\leq p}, \mathbf{mx} \rangle). \tag{4.18}$$

We define $ps_q^{<p}$ analogously.

Algorithm 6 describes how we incorporate cosine similarity bounds within our method. We use the ps bound to index only a few of the non-zeros in each object. Note that, if $ps_q^{<p} < t$, with t defined as in Equation 4.15 and $pd_q^{<p}$, and an object d_c has no features in common with the query in lists I_j, $p \leq j \leq m$, then its cosine similarity to the query will be below t, and its Tanimoto similarity will then be below ϵ. Conversely, if $\left\langle \hat{\mathbf{d}}_q^{>p}, \hat{\mathbf{d}}_c \right\rangle > 0$, the candidate may potentially be a neighbor. By indexing values in each query vector starting at the index p satisfying $ps_q^{\leq p} \geq t$, and then iterating through the index and accumulating, the non-zero values in the accumulator will contain only the suffix dot-products, $\left\langle \hat{\mathbf{d}}_q, \hat{\mathbf{d}}_c^{>} \right\rangle$, where $d_c^{>}$ represents the indexed suffix for the candidate d_c. This portion of the method can be thought of as *candidate generation* (CG), and is similar in scope to the screening phase of many compound search methods in the chemoinformatics literature. Our method uses the un-indexed portion of the candidate,

Algorithm 6 TAPNN with cosine bounds.

function TAPNN-2(D, ϵ)
  $A \leftarrow \varnothing,\ S \leftarrow \varnothing,\ N \leftarrow \varnothing$
  $t \leftarrow 2\epsilon/(1+\epsilon)$
  Compute and store vector lengths for all objects
  Permute objects in non-decreasing vector length order
  for each $q = 1, \ldots, |D|$ s.t. $\|\mathbf{d}_c\| \leq \|\mathbf{d}_q\|\ \forall c \leq q$ **do**
    Normalize d_q
    for each $j = 1, \ldots, m$ s.t. $\hat{d}_{q,j} > 0$ and $ps_q^{\leq p} \geq t$ **do**
      $I_j \leftarrow I_j \cup \{(d_q, \hat{d}_{q,j}, \|\hat{\mathbf{d}}_q^{<j}\|)\}$ ▷ Indexing
    Store $ps_q^{<}$
  for each $q = 1, \ldots, |D|$ s.t. $\|\mathbf{d}_c\| \leq \|\mathbf{d}_q\|\ \forall c \leq q$ **do**
    Find label d_{max} of last object that can be ignored
    for each $j = m, \ldots, 1$ s.t. $\hat{d}_{q,j} > 0$ **do** ▷ CG
      for each $k = S[j], \ldots, |I_j|$ **do**
        $(d_c, d_{c,j}) \leftarrow I_j[k]$
        if $d_c \leq d_{max}$ **then**
          $S[j] \leftarrow S[j] + 1$
        else if $d_c \geq d_q$ **then**
          break
        else if $A[d_c] > 0$ or $ps_q^{\leq j} \geq t$ **then**
          $A[d_c] \leftarrow A[d_c] + \hat{d}_{q,j}\hat{d}_{c,j}$
          Prune if $A[d_c] + \|\hat{\mathbf{d}}_q^{<j}\|\|\hat{\mathbf{d}}_c^{<j}\| < t$
    for each d_c s.t. $A[d_c] > 0$ **do** ▷ CV
      Prune if $A[d_c] + ps_c^{<} < t$
      for each $j = m, \ldots, 1$ s.t. $\hat{d}_{c,j}^{<} > 0$ and $d_{q,j} > 0$ **do**
        $A[d_c] \leftarrow A[d_c] + \hat{d}_{q,j}\hat{d}_{c,j}$
        Prune if $A[d_c] + \|\hat{\mathbf{d}}_q^{<j}\|\|\hat{\mathbf{d}}_c^{<j}\| < t$
      Scale dot-product in $A[d_c]$ according to Equation 4.14
      if $A[d_c] \geq \epsilon$ **then**
        $N \leftarrow N \cup (d_q, d_c, A[d_c])$
  return N

$d_{\hat{c}}^{\leq}$, to complete the dot-product computation during the verification stage, before the scaling and threshold checking steps. We call this portion of the method, which is akin to the verification stage in other chemoinformatics methods, *candidate verification* (CV).

As in `L2AP`, we use a non-increasing inverted list size order for indexing features, which heuristically leads to shorter lists in the inverted index. The partial indexing strategy presented here improves the efficiency of our method in two ways. First, objects

that have non-zero values in common with the query only in the un-indexed set of query features will be automatically ignored. Our method will not encounter such an object in the index when generating candidates for the query and will thus not accumulate a dot-product for it. Second, the verification stage will require reading from memory only those sparse vectors for un-pruned candidates, iterating through fewer non-zeros in general than exist in the un-indexed portion of all objects.

We use the ps bound in two additional ways to improve the pruning effectiveness of our method. First, when encountering a new potential object in the index during the CG stage ($A[d_c] = 0$), we only accept it as a candidate if $ps_q^{\leq j} \geq t$. This is equivalent to the `remscore` bound in `L2AP`. Note that we process index lists in reverse feature processing order in the CG and CV stages. Thus, $A[d_c]$ contains the exact dot-product $\langle \hat{\mathbf{d}}_q, \hat{\mathbf{d}}_c^{>j} \rangle$. Therefore, if $A[d_c] = 0$ and $ps_q^{\leq j} < t$, the candidate cannot be a neighbor of the query object. Second, as a first step in verifying each candidate, we check whether $ps_c^{<}$, added to the accumulated suffix dot-product, meets the threshold t. The value $ps_c^{<}$ is an upper bound of the dot-product of the un-indexed prefix of the candidate vector with any other vector in the dataset. Thus, the candidate can be safely pruned if the check fails.

We check the prefix ℓ^2-norm bound ($l2cg$ and $l2cv$ bounds in `L2AP`) after each accumulation operation, in both the CG and CV stages of the algorithm. The objects cannot be neighbors if the accumulated suffix dot-product, added to the upper bound $\|\hat{\mathbf{d}}_q^{<j}\| \|\hat{\mathbf{d}}_c^{<j}\|$ of their prefix dot-product, cannot meet the threshold t. We have tested a number of additional candidate verification bounds described in the literature based on vector number of non-zeros, prefix lengths, or prefix sums of the vector feature values, but have found them to be less efficient to compute and in general less effective than our described cosine pruning in a variety of datasets. The interested reader may consult [8, 15, 18, 43] for details on additional verification bounds for cosine similarity.

4.5.3 New Tanimoto Similarity Bounds

Up to this point, we have used pruning bounds based on the lengths of the un-normalized vectors and prefix ℓ^2-norms of the normalized vectors to either ignore outright or stop considering (prune) those objects that cannot be neighbors for a given query. We will now present new Tanimoto-specific bounds which combine the two concepts to effect

additional pruning. First, we will describe a bound on the prefix length of an un-normalized candidate vector, which we use during candidate generation. Then, we will introduce a tighter bound for the un-normalized candidate vector length than previously described in [58] which takes advantage of cosine similarity estimates in our method.

A bound on the prefix length of an un-normalized candidate vector

Recall that the dot-product of a query with a candidate vector can be de-composed as the sum of its prefix and suffix dot-products, which can be written as a function of the respective normalized vector dot-products as,

$$\begin{aligned}\left\langle \mathbf{d}_q, \mathbf{d}_c \right\rangle &= \left\langle \mathbf{d}_q^{\leq p}, \mathbf{d}_c \right\rangle + \left\langle \mathbf{d}_q^{>p}, \mathbf{d}_c \right\rangle \\ &= \left\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \right\rangle \|\mathbf{d}_q^{\leq p}\| \|\mathbf{d}_c\| + \left\langle \hat{\mathbf{d}}_q^{>p}, \hat{\mathbf{d}}_c \right\rangle \|\mathbf{d}_q^{>p}\| \|\mathbf{d}_c\|.\end{aligned}$$

For an object that has not yet become a candidate ($A[d_c] = 0$), $\left\langle \hat{\mathbf{d}}_q^{>p}, \hat{\mathbf{d}}_c \right\rangle = 0$, simplifying the expression to,

$$\left\langle \mathbf{d}_q, \mathbf{d}_c \right\rangle = \left\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \right\rangle \|\mathbf{d}_q^{\leq p}\| \|\mathbf{d}_c\|.$$

From the expression $\mathrm{T}(d_c, d_q) \geq \epsilon$, substituting the Tanimoto formula in Equation 2.22, we can derive,

$$\begin{aligned}\left\langle \mathbf{d}_q, \mathbf{d}_c \right\rangle &\geq \frac{\epsilon}{1+\epsilon} \left(\|\mathbf{d}_q\|^2 + \|\mathbf{d}_c\|^2 \right) \\ \|\mathbf{d}_q^{\leq p}\| &\geq \frac{\epsilon}{1+\epsilon} \frac{\|\mathbf{d}_q\|^2 + \|\mathbf{d}_c\|^2}{\|\mathbf{d}_c\| \left\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \right\rangle} \\ \|\mathbf{d}_q^{\leq p}\| &\geq \frac{\epsilon}{1+\epsilon} \frac{\|\mathbf{d}_q\|^2 + \|\mathbf{d}_1\|^2}{\|\mathbf{d}_{q-1}\| \; ps_q^{\leq j}}\end{aligned} \tag{4.19}$$

Equation 4.19 replaces the prefix dot-product $\left\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \right\rangle$ with the ps upper bound, which represents the dot-product of the query with any potential candidate. Furthermore, taking advantage of the pre-defined object processing order in our method, we replace the numerator candidate length by that of the object with minimum length (the first processed object, d_1) and the denominator candidate length with that of the object with maximum length (the last processed object, d_{q-1}). Since $\|\mathbf{d}_1\|^2 \leq \|\mathbf{d}_c\|^2$, $\|\mathbf{d}_{q-1}\| \geq \|\mathbf{d}_c\|$, and $ps_q^{\leq j} \geq \left\langle \hat{\mathbf{d}}_q^{\leq p}, \hat{\mathbf{d}}_c \right\rangle$, the inequality holds.

We use the bound in Equation 4.19 during the candidate generation stage of our method as a potentially more restrictive condition for accepting new candidates. It complements the *ps* bound in line 20 of Algorithm 6, which checks whether new candidates can still be neighbors based only on the prefix of the normalized query vector. Once the prefix length of the query un-normalized vector falls below the bound in Equation 4.19, objects that have not already been encountered in the index can no longer be similar enough to the query.

A tighter bound for the un-normalized candidate vector length

Let $\beta = \|\mathbf{d}_c\|/\|\mathbf{d}_q\|$, and, for notation simplicity, $s = \langle \hat{\mathbf{d}}_q, \hat{\mathbf{d}}_c \rangle = \mathrm{C}(d_i, d_j)$. Given $\mathrm{T}(d_q, d_c) \geq \epsilon$, and the pre-imposed object processing order (i.e. $\|\mathbf{d}_q\| \geq \|\mathbf{d}_c\|$), we derive β as a function of the cosine similarity of the objects,

$$
\begin{aligned}
&\mathrm{T}(d_q, d_c) = \frac{s\|\mathbf{d}_q\|\|\mathbf{d}_c\|}{\|\mathbf{d}_q\|^2 + \|\mathbf{d}_c\|^2 - s\|\mathbf{d}_q\|\|\mathbf{d}_c\|} \geq \epsilon \\
&\epsilon\|\mathbf{d}_c\|^2 - s(1+\epsilon)\|\mathbf{d}_c\|\|\mathbf{d}_q\| - \epsilon\|\mathbf{d}_q\|^2 \leq 0 \\
&\epsilon\beta^2 - (1+\epsilon)s\beta - \epsilon \leq 0 \\
&\beta = \frac{s(1+\epsilon)}{2\epsilon} + \sqrt{\left(\frac{s(1+\epsilon)}{2\epsilon}\right)^2 - 1} = \frac{s}{t} + \sqrt{\left(\frac{s}{t}\right)^2 - 1}
\end{aligned}
\tag{4.20}
$$

Replacing s with any of the upper bounds on the cosine similarity we described in Section 4.5.2, the bound in Equation 4.20 allows us to prune any candidate whose length is less than $\|\mathbf{d}_q\|/\beta$. Note that, for $s = 1$, which is the upper limit of the cosine similarity of non-negative vectors, $\beta = \alpha$, which is the bound introduced by Kryszkiewicz [20] for length-based pruning of candidate vectors. Thus, in the presence of an upper bound estimate of the cosine similarity for two vectors, our bound provides a more accurate estimate of the minimum length a candidate vector must have to potentially be a neighbor for the query.

In Algorithm 7, we present pseudo-code for the `TAPNN` method, which includes all the pruning strategies we described in Section 4.5. The symbol $EQ4.19$ in line 12 refers to checking the query prefix vector length, according to Equation 4.19.

While our bound β for the un-normalized candidate vector length could be checked each time we have a better estimate of the cosine similarity of two vectors, after each

Algorithm 7 The TAPNN algorithm.

1: **function** TAPNN(D, ϵ)
2: Lines 2 – 10 in Algorithm 6
3: **for each** $q = 1, \ldots, |D|$ s.t. $\|\mathbf{d}_c\| \leq \|\mathbf{d}_q\|\ \forall c \leq q$ **do**
4: Find label d_{max} of last object that can be ignored
5: **for each** $j = m, \ldots, 1$ s.t. $\hat{d}_{q,j} > 0$ **do** ▷ CG
6: **for each** $k = S[j], \ldots, |I_j|$ **do**
7: $(d_c, d_{c,j}) \leftarrow I_j[k]$
8: **if** $d_c \leq d_{max}$ **then**
9: $S[j] \leftarrow S[j] + 1$
10: **else if** $d_c \geq d_q$ **then**
11: **break**
12: **else if** $A[d_c] > 0$ or $[ps_q^{\leq j} \geq t$ **and** $EQ4.19]$ **then**
13: $A[d_c] \leftarrow A[d_c] + \hat{d}_{q,j}\hat{d}_{c,j}$
14: **Prune** if $A[d_c] + \|\hat{\mathbf{d}}_q^{<j}\|\|\hat{\mathbf{d}}_c^{<j}\| < t$
15: **for each** d_c s.t. $A[d_c] > 0$ **do** ▷ CV
16: **Prune** if $A[d_c] + ps_c^{<} < t$
17: Compute β given $s = A[d_c] + ps_c^{<}$
18: **Prune** if $\|\mathbf{d}_c\|\beta < \|\mathbf{d}_q\|$
19: Find first j s.t. $\hat{d}_{c,j}^{\leq} > 0$ and $d_{q,j} > 0$
20: $A[d_c] \leftarrow A[d_c] + \hat{d}_{q,j}\hat{d}_{c,j}$
21: **Prune** if $A[d_c] + \|\hat{\mathbf{d}}_q^{<j}\|\|\hat{\mathbf{d}}_c^{<j}\| < t$
22: Compute β given $s = A[d_c] + \|\hat{\mathbf{d}}_q^{<j}\|\|\hat{\mathbf{d}}_c^{<j}\|$
23: **Prune** if $\|\mathbf{d}_c\|\beta < \|\mathbf{d}_q\|$
24: **for each** $j = \ldots, 1$ s.t. $\hat{d}_{c,j}^{\leq} > 0$ and $d_{q,j} > 0$ **do**
25: $A[d_c] \leftarrow A[d_c] + \hat{d}_{q,j}\hat{d}_{c,j}$
26: **Prune** if $A[d_c] + \|\hat{\mathbf{d}}_q^{<j}\|\|\hat{\mathbf{d}}_c^{<j}\| < t$
27: Scale dot-product in $A[d_c]$ according to Equation 4.14
28: **if** $A[d_c] \geq \epsilon$ **then**
29: $N \leftarrow N \cup (d_q, d_c, A[d_c])$
30: **return** N

accumulation operation, it is more expensive to compute than the simpler prefix ℓ^2-norm cosine bound. We thus check it only twice for each candidate object, first after computing the cosine estimate based on the candidate ps bound (line 17), and again after accumulating the first un-indexed feature in the candidate (line 22). We have found this strategy works well in practice.

4.6 Experimental Evaluation for Tanimoto ϵ-NNG Construction

Our Tanimoto ϵ-NNG construction experiment results are organized along several directions. First, we compare the efficiency of our method against existing baselines, demonstrating up to an order of magnitude improvement. Then, we analyze the effectiveness of the new Tanimoto pruning bounds in `TAPNN`, showing that they provide a significant performance benefit. Finally, we analyze the scaling characteristics of our method when dealing with increasing amounts of data.

4.6.1 Baseline Approaches

We compare our methods against the following baselines.

- `IdxJoin` [18] is a straight-forward baseline that does not use any pruning when computing similarities. `IdxJoin` uses an accumulator data structure to simultaneously compute the dot-products of a query object with all other objects, iterating through the inverted lists corresponding to features in the query. While in `L2AP` experiments the method was used to compute dot-products of normalized vectors (see Section 4.4), here we apply the method on the un-normalized vectors. Resulting Tanimoto similarities are computed according to Equation 2.22 using previously stored vector norms. Then, those similarities below ϵ are removed.
- `L2AP` [18] (see Section 4.2) solves the all-pairs problem for the cosine similarity, rather than the Tanimoto similarity. As shown in Section 4.5.2, the Tanimoto all-pairs result is a subset of the cosine all-pairs result. After executing the `L2AP` algorithm, we use Equation 4.14 and previously stored vector norms to compute the Tanimoto similarity of all resulting object pairs and filter out those below ϵ.
- `MMJoin` [15] is a filtering based approach to solving the all-pairs problem for both the cosine and Tanimoto similarities. The Tanimoto solution relies on efficiently solving the cosine similarity all-pairs problem using pruning bounds based on vector lengths and the number of non-zero features in each vector.
- `MK-Join` is an algorithm we designed using the Tanimoto similarity pruning bounds described by Kryszkiewicz in [20] and [58]. `MK-Join` uses an accumulator to compute

similarities of each query against all candidates found in the inverted lists associated with features present in the query. However, MK-Join processes inverted lists in a different order, in non-increasing order of the query feature values. By following this order, Kryszkiewicz has shown in [58] that the method can *safely* stop accepting new candidates once the squared norm of the partially processed query vector (i.e. setting values of unprocessed features to 0) falls below $t = 1 - (\frac{2\epsilon}{1+\epsilon})^2$. A candidate d_c is ignored if its length $\|\mathbf{d}_c\|$ falls outside the range $[(1/\alpha)\|\mathbf{d}_q\|,\ \alpha\|\mathbf{d}_q\|]$, where α is defined as in Equation 4.13.

4.6.2 Execution Efficiency

The main goal of our method is to efficiently solve the Tanimoto APSS problem. We compared TAPNN against the baselines described in Section 4.6.1, for ϵ ranging between 0.6 and 0.99. Figure 4.10 displays our timing results for each method on four datasets. In each quadrant, smaller times indicate better performance. Note that the y-scale has been log-scaled.

The results show that TAPNN significantly outperformed all baselines, by up to an order of magnitude. Speedup of TAPNN versus the next best method ranged between 3.0–8.0x for text datasets, and 1.2–12.5x for chemical datasets. Speedup against IdxJoin, which is similar to a linear search and does not employ any pruning ranged between 8.3–3981.4x for text data and 1.5–519x for chemical data, highlighting the pruning performance of our method, especially for high values of ϵ.

The best performing baseline in general was L2AP, which employs similar cosine based pruning but does not take advantage of un-normalized vector lengths in its filtering. We previously showed that L2AP outperformed MMJoin for the cosine APSS task (see Section 4.4). Our results show that it also outperformed MMJoin for Tanimoto APSS, in all experiments. MK-Join was not competitive against L2AP and MMJoin for $\epsilon \geq 0.8$ for chemical datasets and in general for text datasets. In fact, it performed worse than IdxJoin for the Patents dataset, and only slightly better in general. The Patents dataset has a high average vector size and low average index list size, which may have contributed to the poor performance of MK-Join. The results show that the strategy of cosine filtering applied to the Tanimoto APSS problem, which is employed in different ways by TAPNN, L2AP, and MMJoin, works quite well for both text and chemical

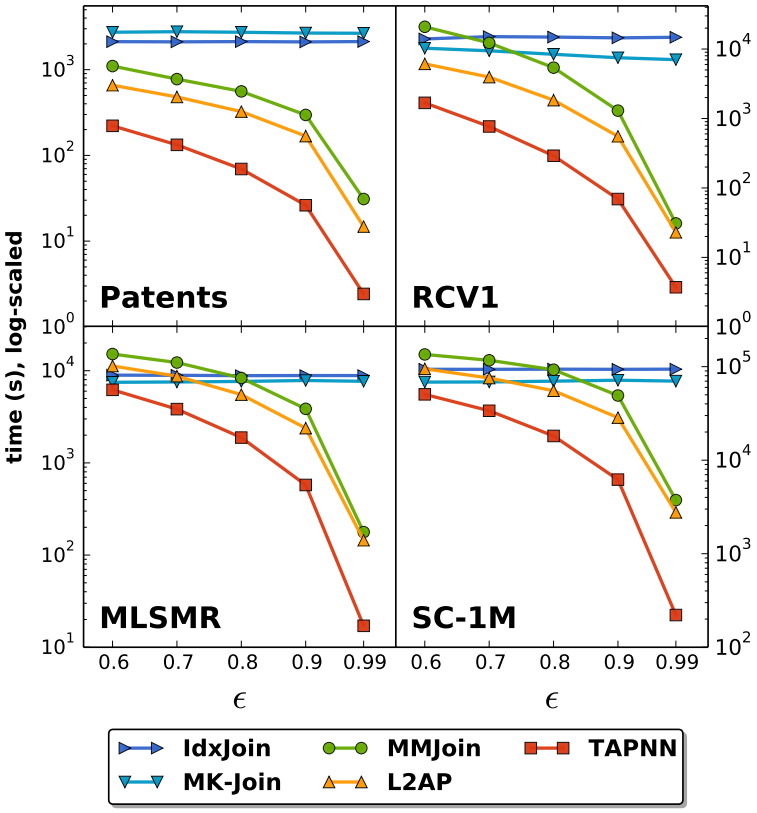

Figure 4.10: Efficiency comparison of TAPNN vs. baselines.

datasets.

4.6.3 Pruning Effectiveness

As a way to test the pruning effectiveness of the new Tanimoto length bounds introduced in Section 4.5.3, we compared execution times of TAPNN with two versions of the program which did not take advantage of these bounds. While both programs implement the length based pruning described in Section 4.5.1, TAPNN-c filters cosine neighbors using the threshold ϵ, while TAPNN-t employs the tighter cosine filtering bound from Equation 4.15. Figure 4.11 shows the outcome of this experiment, displaying the execution times of TAPNN and the the two comparison baselines on four datasets. Note that the execution times are log-scaled. Smaller values indicate better performance.

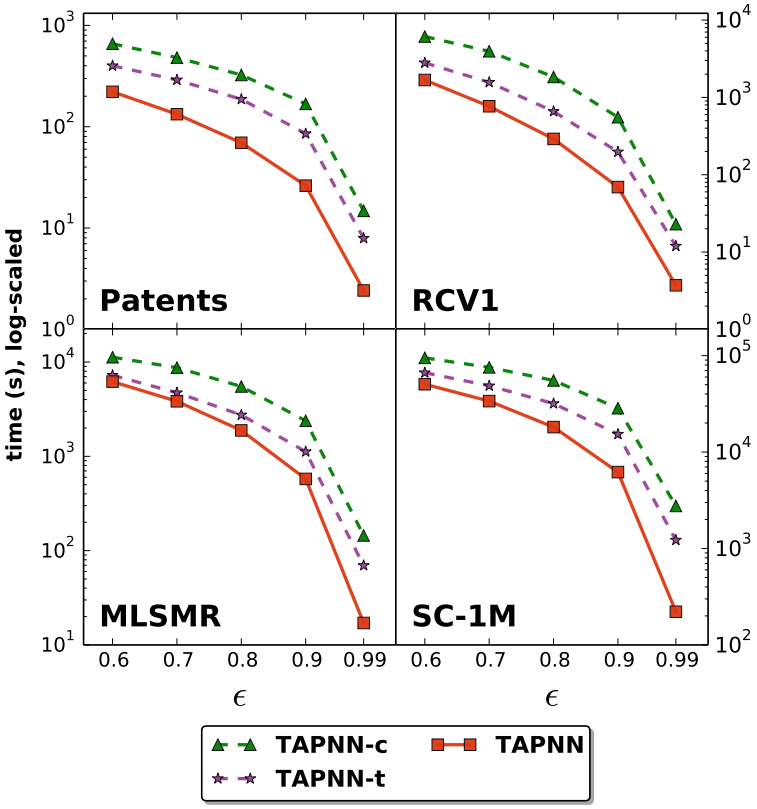

Figure 4.11: Effect of Tanimoto bounds on search efficiency.

The results of our experiments indicate that the newly introduced bounds are effective at improving search performance, achieving up to 5.8x speedup against `TAPNN-t` and 13.3x speedup against `TAPNN-c`. Chemical datasets exhibit higher relative performance improvement at high thresholds, but much lower as $\epsilon \to 0.6$.

4.6.4 Scaling

As a way to understand the scalability of our method and baselines, we executed each method on three random subsets from the SC dataset, containing 100K, 500K, and 1M compounds, respectively, and measuring execution time for ϵ ranging between 0.6 and 0.99. Figure 4.12 (left) shows the results of this experiment. As the problem size was increased, `TAPNN` maintained a similar advantage over the next best alternative, `L2AP`. On the other hand, the performance gap between `L2AP` and `MMJoin`, as well as between

MK-Join and IdxJoin, increased as the problem size was increased.

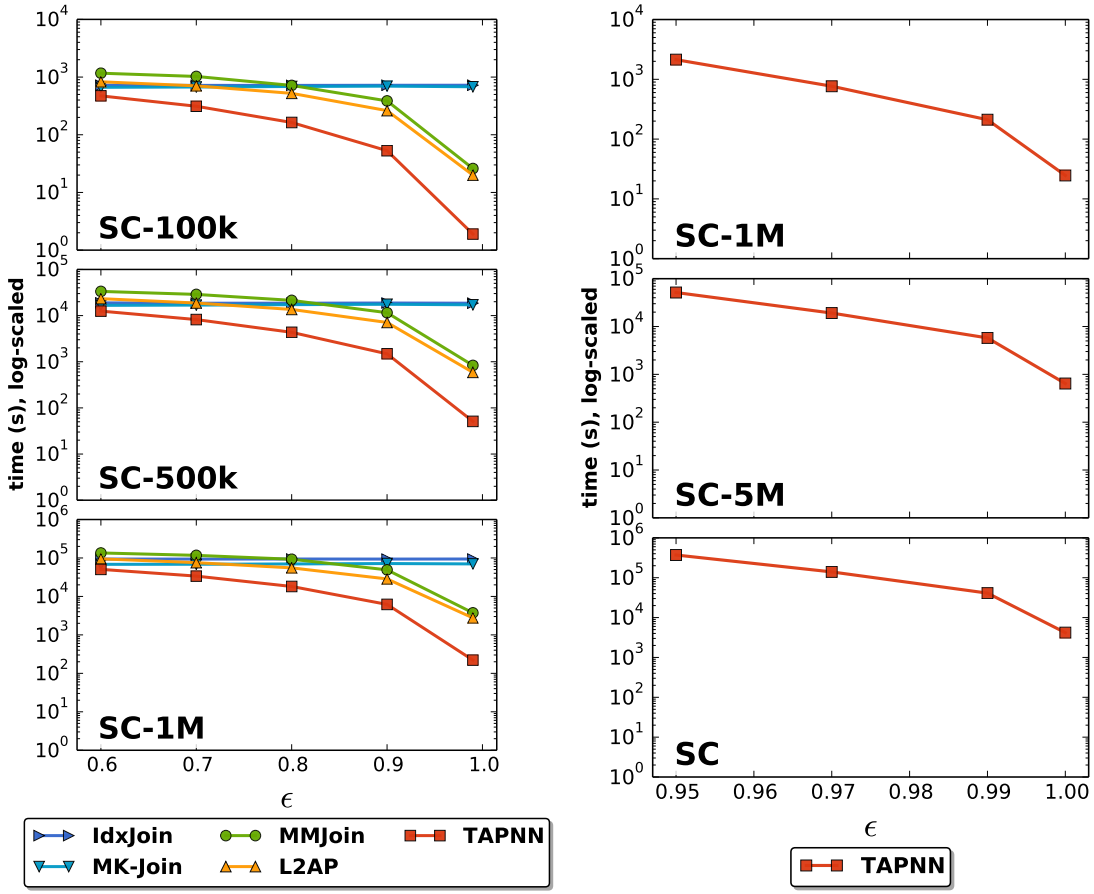

Figure 4.12: Execution time scaling given increasing problem size.

We also tested TAPNN with high similarity thresholds on SC subsets ranging from 500K to 11.5M compounds. Figure 4.12 (right) plots the execution times of TAPNN for datasets with at least 1M compounds. Additionally, Table 4.3 presents results for all four datasets, for $\epsilon \in \{0.95, 0.975, 0.99, 0.999\}$. The columns Δn, Δz, and Δt show relative increases in number of objects, number of non-zeros, and search time, respectively, for corresponding ϵ values, versus the next smaller dataset. For example, the SC-5M dataset has $\Delta n = 5.00$, $\Delta z = 4.52$, and $\Delta t = 26.05$ at $\epsilon = 0.999$, which means that SC-5M has 5x more compounds, 4.52x more non-zeros, and executed 26.05x slower than SC-1M at $\epsilon = 0.999$. The results show a strong correlation between the increase in the

problem size and the search performance in `TAPNN`. Moreover, the relative performance gap was very similar for the different ϵ values, not showing any significant degradation with decreasing ϵ values. As a near-duplicate detection tool, given $\epsilon = 0.999$, `TAPNN` was able to search the entire 11.5M compound SC dataset in a little over an hour, and a 5M subset of the compounds in less than 13 minutes, highlighting its effective pruning and efficient search capabilities.

Table 4.3: Execution time scaling given increasing problem size.

dataset	$\boldsymbol{\epsilon}$	**time (s)**	$\boldsymbol{\Delta}$**n**	$\boldsymbol{\Delta}$**z**	$\boldsymbol{\Delta}$**t**
SC	0.999	4,188.06	2.30	2.55	6.51
SC	0.99	41,099.14	2.30	2.55	7.11
SC	0.975	139,887.44	2.30	2.55	7.32
SC	0.95	(371,520.00)	2.30	2.55	(7.23)
SC-5M	0.999	642.93	5.00	4.52	26.05
SC-5M	0.99	5,778.79	5.00	4.52	27.41
SC-5M	0.975	19,122.77	5.00	4.52	24.96
SC-5M	0.95	51,396.57	5.00	4.52	24.12
SC-1M	0.999	24.68	2.00	2.00	3.78
SC-1M	0.99	210.83	2.00	2.00	4.51
SC-1M	0.975	766.03	2.00	2.00	4.34
SC-1M	0.95	2,130.45	2.00	2.00	4.47
SC-500k	0.999	6.53			
SC-500k	0.99	46.78			
SC-500k	0.975	176.55			
SC-500k	0.95	476.55			

Each section of the table shows ϵ-NNG construction times for a different size subset of the SC dataset, given 4 values of ϵ. The columns Δn, Δz, and Δt show relative increases in number of objects, number of non-zeros, and search time, respectively, for corresponding ϵ values, versus the next smaller dataset in the following section in the table. Note that the time for the SC experiment at $\epsilon = 0.95$ is estimated. The experiment was 95% complete when it was terminated at the end of 4 days (96 hours).

Chapter 5

Serial k-NNG Construction

In this chapter, we introduce `L2Knng` [21], which addresses the *exact* cosine similarity k-NNG construction problem by effectively pruning much of the similarity search space.

5.1 Cosine k-NNG Construction

The `L2Knng` algorithm consists of two distinct steps. In the first step, it uses a fast method that identifies, for each object, k similar objects that may not necessarily be the k nearest neighbors. In the second step, it scans over all the objects and progressively updates the k most similar objects of each object. Specifically, while processing a query object d_q, `L2Knng` updates the k nearest neighbors of all previously processed objects by taking into account their similarity to the query object. At the same time, it updates the k most similar objects of the query object by considering its similarity to the preceding objects. Since the second step potentially considers all pairs of objects, the final set of the k most similar objects for each object are guaranteed to be their k nearest neighbors.

The key to `L2Knng`'s efficiency stems from the following: (i) It uses an index data structure that enables it to quickly find potential neighbors, while pruning some that do not have enough features in common with the object being indexed. (ii) When searching for neighbors, it uses several vector similarity theoretic bounds to filter out many of the potential neighbors found by traversing the index. (iii) It uses a block processing strategy, which leads to efficient traversal of the inverted index lists and additionally improves the effectiveness of the pruning bounds. (iv) Finally, the initial

approximate k-NNG built in its first step is instrumental towards effective indexing and pruning in `L2Knng`.

5.1.1 Approximate Graph Construction

`L2Knng-a` is the inexact k-NNG construction method used by `L2Knng` to build an initial approximate graph. It consists of two steps. First, it builds a set of initial neighborhoods, relying on the idea that *high-weight features count heavily towards the similarity of two vectors* [37, 84]. Then, given that *an object's neighbor's neighbor is also likely their neighbor* [36, 85], it iteratively enhances the k-NNG by looking for new candidates in each neighbor's neighborhood.

The first step is achieved as follows. For each object d_i, `L2Knng-a` builds a list of up to μ ($\mu \geq k$) candidates, choosing among those objects that have features in common with d_i until there are no more features to check or μ candidates were found. It then computes the exact similarities of all candidates with d_i and adds the objects with the top k values to d_i's initial neighborhood.

The choice of candidate objects is crucial to obtaining an approximate graph that is close to the exact k-NNG. `L2Knng-a` pre-processes input vectors to have unit length and uses an inverted index to identify candidate objects with common features with d_i. As a heuristic way to prioritize high-weight common features, it sorts the features in each vector in decreasing weight order, and sorts each of the lists in the inverted index in decreasing order of feature weights. `L2Knng-a` then traverses two index lists at a time, in decreasing order of their associated weights in $\mathbf{d}_i$. From the two lists, it chooses the candidate d_c with the higher prefix dot product, which is more likely to be a true neighbor.

In the second step, `L2Knng-a` executes up to γ iterative neighborhood enhancement updates, in which, for each object d_i, its current neighborhood is updated by taking into account its similarity to some of the objects that are neighbors of its neighbors. This is done as follows. `L2Knng-a` traverses d_i's neighborhood in decreasing order of its neighbor similarities. Given some neighbor d_j, it then traverses its neighborhood, in decreasing order of d_j's neighbor similarities, to identify potential neighbors for d_i. Avoiding objects that are already in d_i's neighborhood or have d_i in their neighborhood, `L2Knng-a` greedily chooses as candidates only those neighbor's neighbors d_k with a similarity value greater

or equal than that between the query vector and its neighbor, $\text{sim}(\mathbf{d}_j, \mathbf{d}_k) \geq \text{sim}(\mathbf{d}_i, \mathbf{d}_j)$, and limits the size of the candidate list to be μ. `L2Knng-a` then computes similarities between $\mathbf{d}_i$ and candidates, updating both relevant neighborhoods with the results. We use δ as an early termination parameter, stopping iterations early if less than $\delta k|D|$ neighborhood changes occurred in an update.

Our strategy for choosing candidates in the first step improves upon the work of Park et al. [37] by limiting, for each object, the number of computed similarities. High quality candidates are greedily chosen from few inverted index lists. The neighborhood enhancement step improves upon the work of Dong et al. [36] in two ways. First, it ensures an upper bound on the number of similarity computations and prioritizes those candidates more likely to improve the neighborhood. Second, the enhancement steps will probably converge faster and to higher recall, as the input neighbors likely have higher similarity values than the randomly chosen neighbors in their method.

5.1.2 Filtering

`L2Knng` uses a similar filtering framework as the one described for `L2AP` in Section 4.2. However, there is a key difference between the filtering solutions to the two problems. ϵ-NNG construction seeks to prune object pairs with a similarity below a threshold ϵ, while `L2Knng` filters those pairs that cannot improve k-neighborhoods. These distinct goals lead to very different pruning bounds in the two methods. The threshold ϵ is an input to the ϵ-NNG construction problem. In the k-NNG construction problem, ϵ could be chosen to be the minimum neighborhood similarity σ_{d_i} among all objects in the true k-NNG, which is unknown and would nonetheless be a suboptimal filtering choice while constructing the k-NNG. Instead, we devise better bounds, detailed in the remainder of this section, that can be used in each stage of the method to *safely* prune object pairs that cannot be a part of the true k-NNG.

Indexing

`L2Knng` uses information in the approximate k-NNG to prune some object pairs by indexing only a subset of the features in each object. In each iteration, `L2Knng` needs to identify among the previously processed objects those whose neighborhoods can be updated by including the *query* object d_q. Similar to `L2AP`, `L2Knng` builds the index

incrementally, delaying the indexing of d_q until after its processing. Future potential neighbors can only improve d_q's neighborhood if their similarity with d_q is higher than the minimum similarity in d_q's neighborhood, σ_{d_q}. Therefore, `L2Knng` indexes objects until their suffix ℓ_2-norm falls below its minimum neighborhood similarity σ_{d_q}[1] As shown in Section 4.2.1, given that all vectors in the dataset have unit length, based on the Cauchy-Schwarz inequality, the suffix ℓ_2-norm of d_q is an upper bound of the similarity of d_q's suffix with any other object, including unprocessed objects in the set,

$$\left\langle \mathbf{d}_q^{>j}, \cdot \right\rangle \leq \|\mathbf{d}_q^{>j}\| \|\cdot\| \leq \|\mathbf{d}_q^{>j}\|.$$

The object processing order in `L2Knng` differs from the one in `L2AP`. Note that, since indexing occurs after finding neighbors for an object, the query object d_q must also be identified when processing future objects if d_q can improve their neighborhoods. `L2Knng` thus fixes the object processing order based on the minimum similarities in the initial approximate k-NNG it builds before processing objects. To see why this is necessary, consider the following scenario. Let ϵ_{d_i} be the minimum neighborhood similarity for some object d_i at the time of its indexing (*indexing threshold*), which later may be different than σ_{d_i}. Consider indexing the object d_i at a threshold ϵ_{d_i} and then processing an object d_j with a smaller minimum neighborhood similarity. If $\sigma_{d_j} \leq \text{sim}(\mathbf{d}_i, \mathbf{d}_j) < \epsilon_{d_i}$, then d_i is no longer guaranteed to be found when processing d_j, and d_j's neighborhood may be inexact at the end of the algorithm execution. Therefore, to ensure correctness, objects must be indexed in a strictly non-decreasing indexing threshold order.

Algorithm 8 Indexing in L2Knng

```
1: function INDEX(d_q, I, se, ε_{d_q})
2:     b ← 1
3:     for each j = 1, ..., m, s.t. d_{q,j} > 0 and √b ≥ ε_{d_q} do
4:         b ← b − d_{q,j} d_{q,j}
5:         I_j ← I_j ∪ {(d_q, d_{q,j}, ‖d_q^{>j}‖)}
6:     se[d_q] ← ‖d_q^{>j}‖
```

[1] A keen observer will note that `L2Knng` indexes the prefix while `L2AP` indexes the suffix of each object. Note also that the feature processing order in the candidate generation and verification stages of the algorithm, as well as the initial feature sorting are reversed. We noted in Section 4.2 that other methods, such as `MMJoin` [15] also follow this feature processing order. This does not affect the pruning effectiveness of the bounds and is merely an implementation choice.

Algorithm 8 details the indexing procedure in `L2Knng`. The prefix of a vector $\mathbf{d}_q$ is indexed while its suffix ℓ_2-norm, computed in b, is above or equal to our threshold ϵ_{d_q} (lines 3-5). The suffix ℓ_2-norm at each indexed feature (line 5) and the suffix ℓ_2-norm of the un-indexed portion of $\mathbf{d}_q$ (*suffix estimate*, line 6) are also stored, to be used in other stages of the algorithm. We denote by $\mathbf{d}_q^{\leq}$ the indexed prefix of object d_q and by $\mathbf{d}_q^{>}$ its un-indexed suffix.

Pruning the Search Space

A computed similarity can only improve the neighborhood of a query object d_q if it is above σ_{d_q}. Furthermore, it can only improve neighborhoods of already processed objects if it is greater than the minimum of all neighborhood similarities of indexed objects. To keep track of this value, `L2Knng` could update a heap data structure each time the neighborhood of an indexed object is improved, but we have found this affects overall efficiency. Instead, `L2Knng` approximates this value by the minimum indexing threshold among all indexed objects, denoted by it, which is strictly smaller than the current minimum of all indexed objects' neighborhood similarities. Using a similar idea as during indexing, `L2Knng` only starts accumulating for an object d_c while the query suffix ℓ_2-norm is above the lower of these two bounds, $\min(it, \sigma_{d_q})$. Once the suffix ℓ_2-norm falls below this threshold, only index values for objects with non-zero accumulated partial dot-products are processed. Additionally, `L2Knng` uses the initial approximate k-NNG and the current version of the k-NNG to bypass already computed similarities.

During both the candidate generation and verification stages, there is a further opportunity for pruning when a common feature j is encountered between the query and candidate vectors. To be useful, the final similarity value should improve the neighborhoods of either the query or candidate objects. The accumulator contains the exact similarity of the two prefix vectors, and the similarity of the suffix vectors can be estimated, based on the Cauchy-Schwarz inequality, as upper bounded by the product of their suffix ℓ_2-norms [18]. Thus, a candidate can be pruned if

$$A[d_c] + \|\mathbf{d}_q^{>j}\|\|\mathbf{d}_c^{>j}\| < \min(\sigma_{d_q}, \sigma_{d_c}).$$

`L2Knng` employs one additional pruning strategy during the candidate verification

Algorithm 9 Searching for neighbors in L2Knng.

1: **function** FINDNEIGHBORS($\mathbf{d}_q, \mathcal{I}, se, it, \hat{\mathcal{N}}, \mathcal{N}$)
2: $r \leftarrow 1$; $A \leftarrow \varnothing$ ▷ accumulator
3: $A[d_c] \leftarrow \emptyset$ for neighbors d_c in $\hat{\mathcal{N}}$ and $\mathcal{N}$
4: **for each** $j = 1, \ldots, m$, s.t. $d_{q,j} > 0$ **do** ▷ Candidate Generation
5: **for each** $(d_c, d_{c,j}, \|\mathbf{d}_c^{>j}\|) \in I_j$ **do**
6: **if** $A[d_c] > 0$ **or**
$[A[d_c] \neq \emptyset$ **and** $\sqrt{r} \geq \min(it, \sigma_{d_q})]$ **then**
7: $A[d_c] \leftarrow A[d_c] + d_{q,j} d_{c,j}$
8: **if** $A[d_c] + \|\mathbf{d}_q^{>j}\| \|\mathbf{d}_c^{>j}\| < \min(\sigma_{d_q}, \sigma_{d_c})$
then
9: $A[d_c] \leftarrow \emptyset$
10: $r \leftarrow r - d_{q,j} d_{q,j}$
11: **for each** d_c s.t. $A[d_c] > 0$ **do** ▷ Candidate Verification
12: **next** d_c **if** $A[d_c] + se[d_c] < \min(\sigma_{d_q}, \sigma_{d_c})$
13: **for each** j s.t. $d_{c,j}^{>} > 0 \wedge d_{q,j} > 0$ **do**
14: $A[d_c] \leftarrow A[d_c] + d_{q,j} d_{c,j}$
15: **if** $A[d_c] + \|\mathbf{d}_q^{>j}\| \|\mathbf{d}_c^{>j}\| < \min(\sigma_{d_q}, \sigma_{d_c})$ **then**
16: **next** d_c
17: $N_{d_c} \leftarrow N_{d_c} \cup \{(d_q, A[d_c])\}$ **if** $A[d_c] > \sigma_{d_c}$
18: $N_{d_q} \leftarrow N_{d_q} \cup \{(d_c, A[d_c])\}$ **if** $A[d_c] > \sigma_{d_q}$

stage. The $se[d_c]$ suffix estimate value that was stored when indexing the candidate d_c estimates the dot-product between the un-indexed portion of d_c and any other vector in the dataset, $\text{sim}(\mathbf{d}_c^{>}, \cdot)$. We use this value here as an estimate for the similarity between the query and candidate suffix, $\langle \mathbf{d}_q, \mathbf{d}_c^{>} \rangle$. If the sum of the accumulated score and the estimate falls below $\min(\sigma_{d_q}, \sigma_{d_c})$, the candidate is discarded.

Having presented the different pruning bounds used in `L2Knng`, note that their effectiveness would be greatly reduced without first computing the initial approximate k-NNG. First, indexing thresholds for unprocessed objects would be unknown, and `L2Knng` would have to index all object features, missing an important pruning opportunity. Similarly, during a search, the algorithm would have to consider all possible candidates with common features, as the minimum indexing threshold it would be 0. Finally, the minimum neighborhood similarities of previously processed objects would likely be smaller, leading to less object pairs being pruned and more neighborhood updates. While `L2Knng` does not require the initial approximate graph to be computed by `L2Knng-a`, an initial graph with high recall will lead to more effective pruning and higher efficiency in constructing the exact graph.

Algorithm 9 delineates the procedure used to find neighbors in `L2Knng`. The variable r computes the suffix ℓ_2-norm of the query vector, which is used to prevent accumulating similarity for objects that cannot improve neighborhoods (line 6) in the candidate generation stage. At the end of the verification stage, the accumulator contains the exact similarity between the query and objects that survive pruning. These objects are added to the candidate or query neighborhoods if they can improve them.

5.1.3 Block Processing

Algorithm 10 The L2Knng algorithm.

```
1: function L2Knng(D, k, μ, γ, δ, ν)
2:   N̂ ← L2Knng-a (D, k, μ, γ, δ)
3:   Reorder dimensions in non-decreasing frequency order
4:   N ← N̂;
5:   for each i = 1, 2, ..., n do
6:     ε_{d_q} ← σ_{d_q}, it ← min(it, ε_{d_q})
7:   I_j ← ∅, Z ← Z ∪ I_j, for j = 1, ..., m
8:   for each i = 1, ..., n s.t. ε_{d_q} ≤ ε_{d_j}, ∀j > i
       do
9:     FindNeighbors(d_q, Z, se, it, N̂, N)
10:    Index(d_q, Z, se, ε_{d_q})
11:    if i % |D|/ν = 0 then
12:      it ← CompleteBlock(i, Z, se, N̂, N, ν)
13: return N
```

Algorithm 10 gives an overview of `L2Knng`. The initial approximate graph k-NNG ($\hat{\mathcal{N}}$, line 2) bootstraps the search framework, providing the necessary processing order for the main loop. As suggested by Bayardo et al. [8], we reorder dimensions in all vectors in non-decreasing object frequency order as a heuristic way to minimize the inverted index size.

The index keeps growing as more and more objects are processed. The minimum indexing threshold it defined by the initial k-NNG is likely very small, causing the majority of objects in the index to become candidates for each subsequent query. While many candidates will later be eliminated based on pruning bounds that take advantage of continuously updated neighborhood similarities, the delayed pruning can lead to slower execution. `L2Knng` improves the indexing threshold by periodically "flushing" the index. After completing the k-NNG construction for the already indexed objects, the index

can be discarded, speeding up future candidate generation and providing an improved minimum indexing threshold it. Neighborhood construction can be finalized for a block of objects by executing *FindNeighbors* for all un-processed vectors, without indexing them. `L2Knng` then uses the updated minimum neighborhood similarities of unprocessed objects to define a new processing order. Given a number of blocks parameter ν, `L2Knng` finalizes a block of indexed objects after processing every $|D|/\nu$ objects.

5.2 Experimental Evaluation for Cosine k-NNG Construction

Our cosine k-NNG construction experiment results are organized along two directions. First, we test `L2Knng-a` against approximate baselines, comparing recall and execution times given different candidate list sizes, and testing their efficiency returning a neighborhood graph that is at least 95% accurate. Second, we evaluate the runtime and memory scalability of `L2Knng` as the number of input objects increases, and its efficiency as opposed to exact baselines.

5.2.1 Baseline Approaches

We compare our methods against the following baselines.

- `kIdxJoin` is a straight-forward baseline similar to *IDX* in [37] that first builds a full inverted index. Then, without performing any pruning, it uses the index to compute exactly, via accumulation, the similarity of each object with all other objects in the set, returning the top-k matches for each query object.
- `kL2AP` solves the k-NNG problem by executing similarity searches using `L2AP` [18]. We modified `L2AP` to allow specifying a set of input query vectors. Then, as we iteratively reduce the search threshold ϵ, we provide as input only those objects with incomplete neighborhoods.
- *BMM* refers to the docID-oriented with variable block sizes version of the Block-Max Maxscore method by Dimopoulos et al. [62]. The method splits inverted lists into blocks and uses maximum scores for postings in each block to prune the similarity search space. We adapted the method for cosine similarity ranking and chose the

same block sizes as in their paper. Blocks were stored in compressed form, using PForDelta compression [86].

- *Maxscore* is an in-memory implementation of the *max_score* information retrieval algorithm [64], as described by Dimopoulos et al. in [62], adapted to rank based on cosine similarity.
- *Greedy Filtering* is a state-of-the-art approach for solving the approximate k-NNG construction problem applied to sparse weighted vectors, proposed by Park et al. [37].
- *NN-Descent* was designed by Dong et al. [36] to work with generic similarity measures and has been shown effective at solving the approximate k-NNG construction problem in both sparse and dense datasets.

While LSH has been a popular method for top-k search, it does not perform well in the k-NNG construction setting. Both *Greedy Filtering* and *NN-Descent* have been shown to outperform LSH when applied to this problem, for k typically ≥ 10. Additionally, `L2AP` outperformed LSH in the related APSS problem. As we will show in Section 5.2.3, `L2Knng` significantly outperforms `kL2AP`, the k-NNG method based on `L2AP`, as well as *Greedy Filtering* and *NN-Descent*. As a result, we have chosen not to compare against LSH in this work.

5.2.2 Evaluation of Approximate Methods

Candidate pool size parameter analysis

The efficiency of all the approximate methods under consideration are dependent on the number of candidates they are allowed to consider for each object, μ. The larger the candidate pool is, the more likely the true neighborhood is found among the objects in the pool. We compare the recall and execution time of `L2Knng-a` with other approximate baselines, given the same candidate list and neighborhood size parameters, μ and k. We tested each method, without changing any other parameters, given $\mu = k, 2k, \ldots, 10k$, on the RCV1-400k and WW200-250k datasets. We tested `L2Knng-a` with $\gamma = 0$ (`L2Knng-a`$_0$), which does not execute any iterative neighborhood updates, and with $\gamma = 3$ (`L2Knng-a`$_3$).

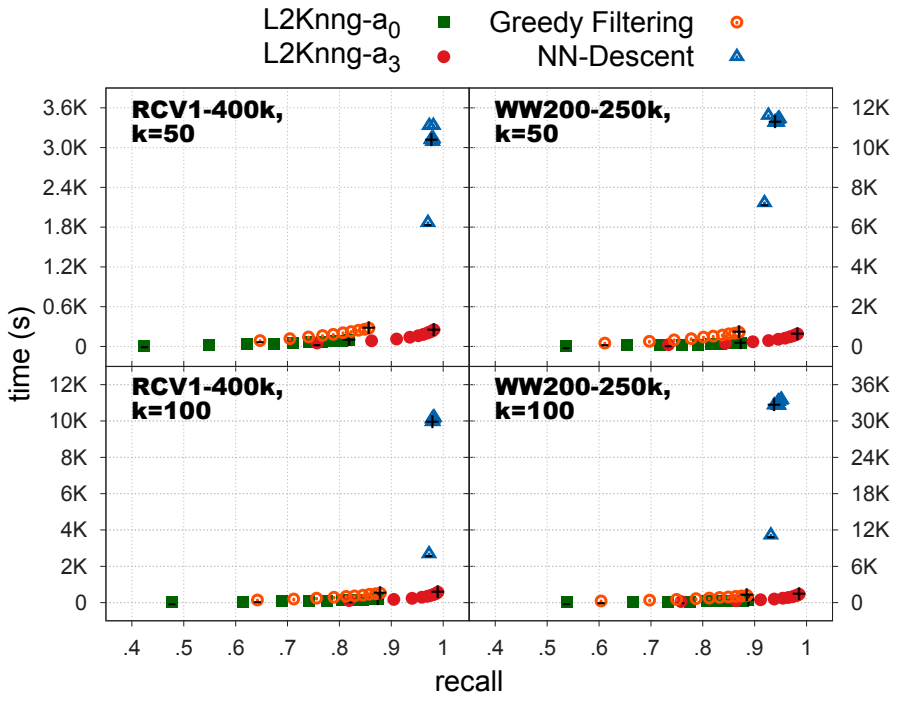

Figure 5.1: Recall and execution time of approximate methods given increasing candidate pool sizes.

Figures 5.1 and 5.2[2] plot recall versus execution time for our experiment results. Figure 5.1 on the left contains all the results. For a better comparison of the remaining methods, Figure 5.2 leaves out results for the *NN-Descent* method. For all methods, results for $\mu = k$ are marked with a "-" label, and those for $\mu = 10k$ with a "+" label. The best results are those points in the lower-right corner of each quadrant in the figure, achieving high recall in a short amount of time. We display results for $k \in \{50, 100\}$. Results for other k values showed similar trends.

Methods generally exhibit higher recall and higher execution time for larger μ values. *NN-Descent* took considerably more time than all the other methods to complete the graph construction. `L2Knng-a`$_0$ takes much less time to execute than *Greedy Filtering*

[2]The experiment results we published in [21] contained results for the *NN-Descent* baseline that were accidentally executed in parallel, with 8 threads, instead of serially. Since our methods and all other baselines are serial programs, we have re-executed the affected experiments, using the same version of the *NN-Descent* library (v. 1.2) and on the same computing environment as we used for the experiments in [21].

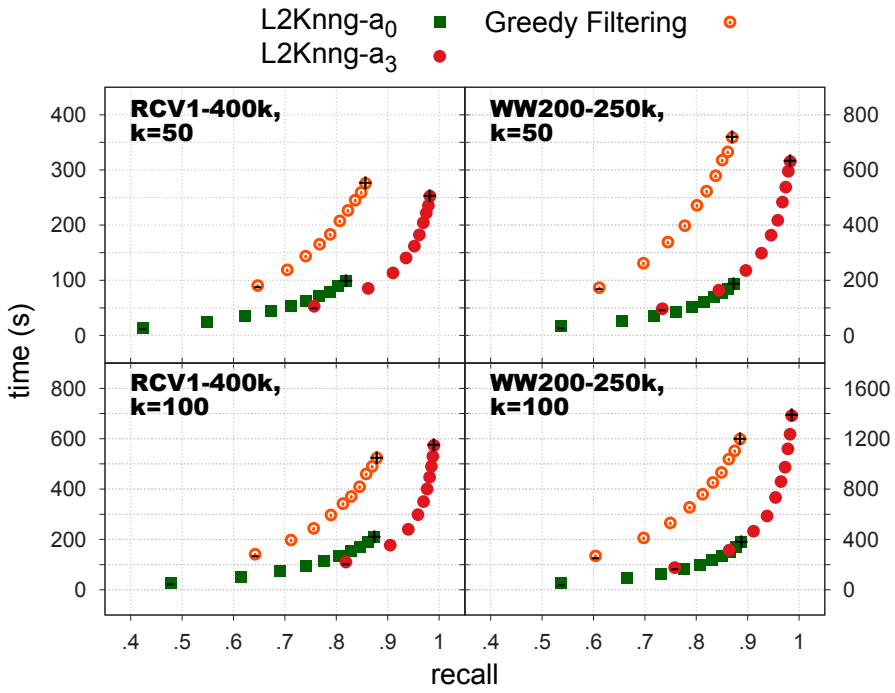

Figure 5.2: Recall and execution time of approximate methods except *NN-Descent* given increasing candidate pool sizes.

and, given large enough μ, can achieve similar or higher recall. Both `L2Knng-a` and *Greedy Filtering* require larger μ values than *NN-Descent* to achieve high recall. Yet, *NN-Descent* does not improve much as μ increases. `L2Knng-a`$_3$ is able to outperform both competitors, with regards to both time and recall, for large enough μ.

L2Knng-a efficiency

In this work, we focused on building the exact k-NNG. While approximate methods cannot easily achieve perfect recall, we compared their efficiency when seeking a close approximation of the true k-NNG. We executed each approximate method under a wide range of parameters and report the smallest time for which a minimum recall value of 0.95 was achieved. Figure 5.3 presents execution times for the approximate methods, for four of the datasets. Results for the other datasets were similar. We also include the times for our exact variant, `L2Knng`, as comparison. We were not able to achieve high enough recall for NN-Descent for the WW200 dataset. As such, *NN-Descent* results are

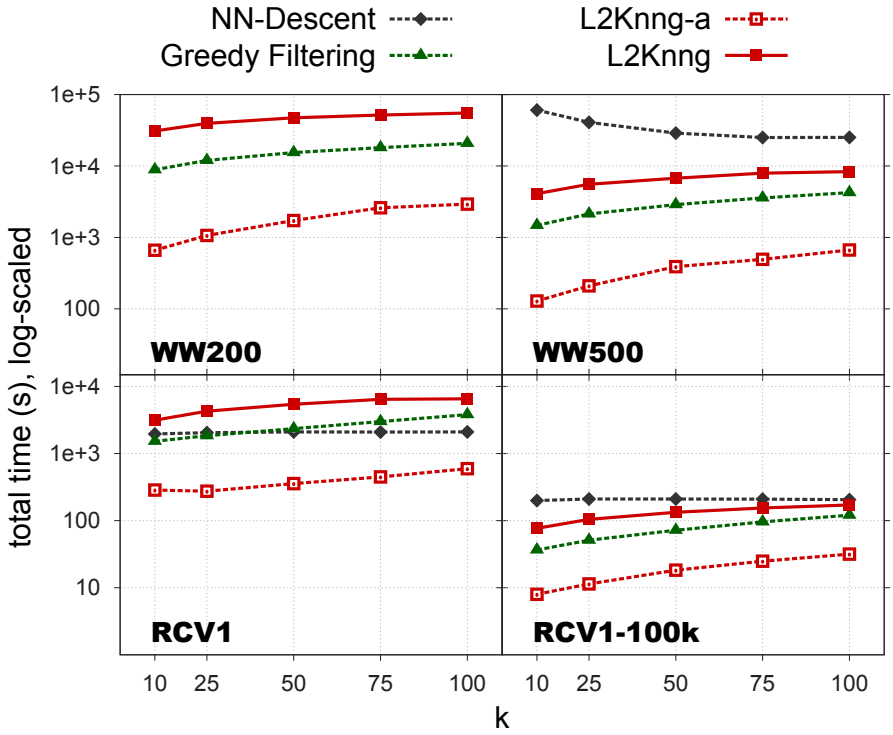

Figure 5.3: Approximate k-NNG construction efficiency.

not included in the upper left quadrant of the figure. In each quadrant of the figure, smaller values represent better results. Note that the total time values (y-axis) are log-scaled.

`L2Knng-a` was more efficient than all baselines in all experiments. For problems where perfect recall is not needed, `L2Knng-a` can provide a close approximation in much less time. *NN-Descent* performed poorly on the WW datasets. This may be explained by the much higher dimensionality and mean row length of the WW datasets as compared to the RCV1 datasets, which can lead to repeated inclusion of objects in computationally expensive *NN-Descent* local joins. In contrast, `L2Knng-a` uses several strategies that limit the number of computed dot-products. It builds a higher quality initial graph than *NN-Descent*, prioritizes candidate inclusion, and sets a hard limit on the candidate list size in each iterative update.

5.2.3 Evaluation of Exact Methods

Initial graph influence

While the final recall for an `L2Knng` execution is 1.0, our method uses an initial approximate graph $\hat{\mathcal{N}}$ as a guide in its k-NN search. A graph $\hat{\mathcal{N}}$ with high recall provides `L2Knng` with higher minimum neighborhood similarity values, which translate into tighter pruning bounds and leads to fewer full vector dot-products being computed (smaller scan rate) and faster runtime. We tested the influence of the initial graph quality in three scenarios on the RCV1-400k and WW200-250k datasets. In the first scenario (*random*), we generated an initial graph by randomly picking k neighbors for each object from the set of objects with which they shared at least one feature in common. In the second scenario (*fast*), we chose parameters that ensure fast execution, without guaranteeing high recall ($\mu = k,\ \gamma = 1$). We executed the search using 10 completion blocks in both scenarios ($\nu = 10$). Finally, we include for comparison the best results we achieved after a parameter search (*best*).

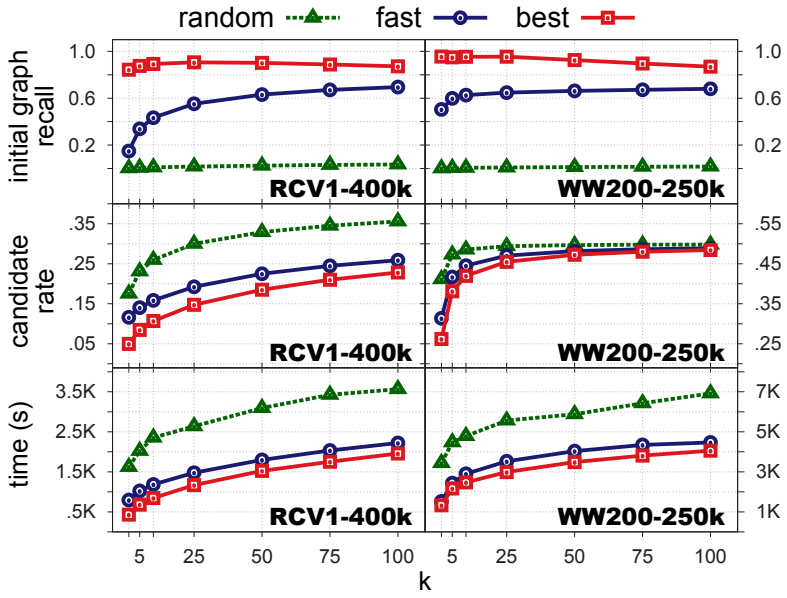

Figure 5.4: Initial graph influence over `L2Knng` efficiency.

Figure 5.4 presents our experiment results. The top of the figure shows, for each k value, the recall of the initial graph for the two tested datasets. The middle and

bottom of the figure show, for each k value, the candidate rate and execution time after completing the *exact* k-NNG construction. The results emphasize the importance of the initial graph quality in `L2Knng`. The initial graph $\hat{\mathcal{N}}$ for the *random* test case had recall 0.018 and 0.009 on average across all k values for the RCV1-400k and WW200-250k datasets, respectively. The recall was 0.496 and 0.628 in the *fast* and 0.883 and 0.929 in the *best* test cases. These better initial graphs translate into both lower candidate rates and smaller execution times. The *fast* case performed similarly to the *best* case, showing that `L2Knng` can be used with reasonable parameter values and does not require extensive parameter tuning.

Parameter sensitivity

The parameters μ, γ, and ν can influence the effectiveness and efficiency of our exact and approximate algorithms. Larger values for μ increase the number of candidates considered for building the initial graph and will likely lead to increased recall for this stage. Similarly, higher γ values translate to more iterations of initial neighborhood enhancement, at the cost of more similarity computations. Increasing ν values can lead to improved candidate generation pruning and faster index traversal, at the cost of reading vectors in unprocessed blocks several times to find similarities with indexed vectors. There is a trade-off between the benefit of more efficacious pruning bounds and the time taken to achieve them.

We executed parameter sensitivity experiments on the RCV1-400k and WW200-250k datasets, for $k \in \{25, 50, 75, 100\}$. In each experiment, we fixed two of the parameters and varied the third. In the first experiment, given $\gamma = 0$, and $\nu = 10$, we varied μ between 100 and 1000. In the second experiment, given $\mu = 300$, and $\nu = 10$, we varied γ between 0 (no initial neighborhood enhancement) and 10. Finally, to verify the sensitivity of the number of blocks parameter, ν, given $\mu = 300$, and $\gamma = 1$, we varied ν between 1 and 500.

Figures 5.5 and 5.6 show the results of our experiments testing the sensitivity of `L2Knng` to different values of μ and γ. As expected, the recall value grows when either μ or γ are increased. While the rise is sharp at first, it levels off quickly as the parameter values get larger, showing that the most benefit is gained from checking a relatively small number of initial candidates and executing few rounds of initial neighborhood

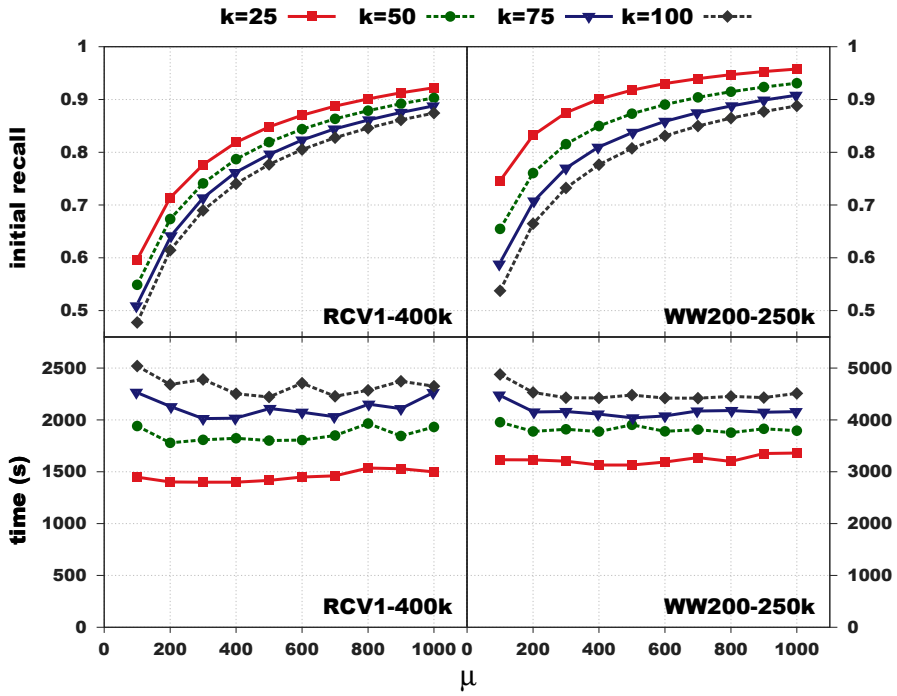

Figure 5.5: `L2Knng` μ parameter sensitivity.

enhancement. In general, the best results we obtained after parameter tuning were executed with $1 \leq \gamma \leq 3$ and $300 \leq \mu \leq 500$. The execution time was not greatly affected as we increased the values of μ or γ, showing that `L2Knng` is not very sensitive to these parameter choices.

Figure 5.7 shows the execution times from experiments measuring the sensitivity of `L2Knng` to ν. The results show that increasing the number of blocks ν initially leads to improved performance for all k values. While the improvement is more drastic at first, ν values greater than 50 do not improve the results much, and can eventually lead to decreased efficiency.

Pruning effectiveness

`L2Knng` works by pruning the majority of the candidates that are not true neighbors. Candidates can be pruned while checking the suffix ℓ_2-norm at a common feature during the candidate generation stage (cg), during the candidate verification stage (cv), or after checking the suffix estimate score (ses). Since a partial dot-product is accumulated

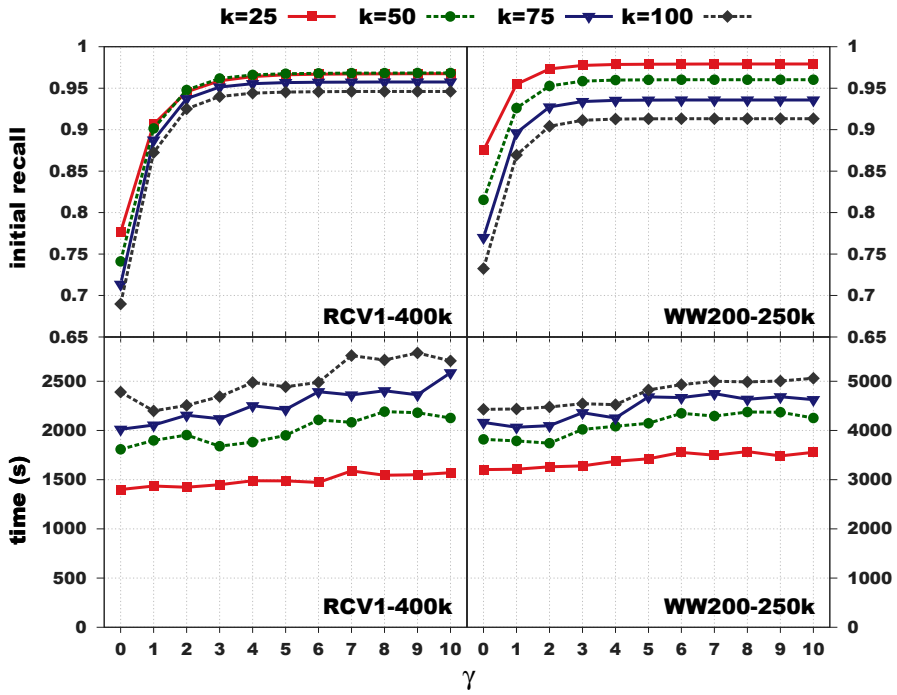

Figure 5.6: `L2Knng` γ parameter sensitivity.

for each candidate before being pruned, it is important that candidates be pruned as early as possible. In an experiment in which we used fast defaults for all parameters ($\mu = k,\ \gamma = 1,\ \nu = 10$), we counted the number of candidates that were pruned in each stage of the algorithm. Additionally, we display the number of candidates that survived all pruning and had full dot-products computed (*dps*). Figure 5.8 shows the results of this experiment for the RCV1-400k and WW200-250k datasets, as stacked bar charts showing the number of candidates for each category.

Results show that the majority of objects are pruned soon after becoming candidates, in the candidate generation stage (*cg*). Of the remainder, most are pruned by the suffix estimate bound (*ses*), which is checked once, at the beginning of the candidate verification stage, and by additional pruning in the candidate verification stage (*cv*). On average, across all k values, 0.15% and 0.02% of candidates survived all pruning for the RCV1-400k and WW200-250k datasets, respectively. A large number of objects never become candidates in `L2Knng`, as a result of either the ℓ_2-norm based candidate acceptance bound in the candidate generation stage of the algorithm, or due to the

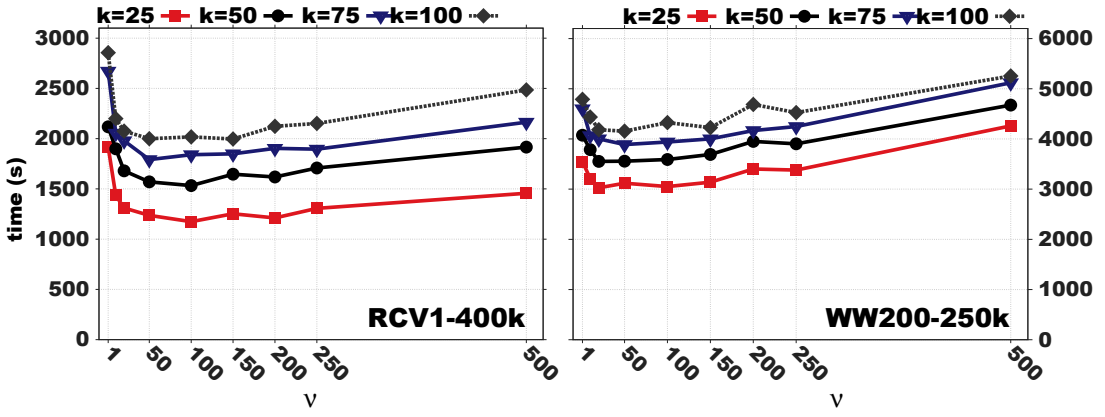

Figure 5.7: `L2Knng` ν parameter sensitivity.

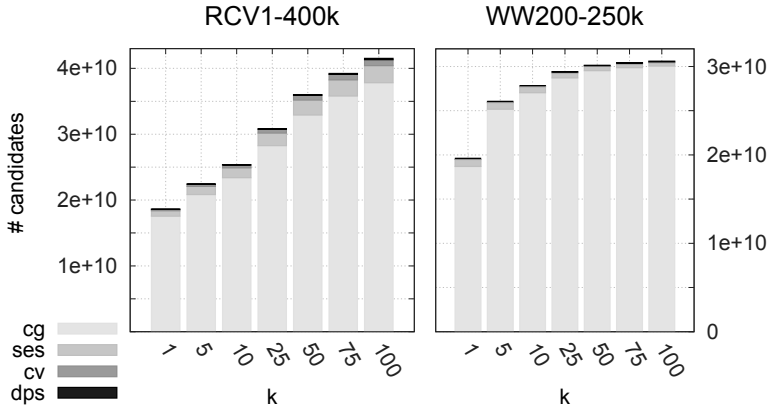

Figure 5.8: Candidate pruning in `L2Knng`.

prefix-filtering based index reduction. On average across all k values, only 38.17% and 88.66% of all potential candidates actually became candidates for the RCV1-400k and WW200-250k datasets.

Scalability testing

As the dataset size increases, the exact k-NNG problem will take longer to solve, as each object has more potential neighbors that have to be vetted. As a way to verify scalability, we tested our methods on three subsets of the RCV1 and two subsets of the WW200 datasets. For each data subset, Table 5.1 reports, for $k \in \{25, 50, 75, 100\}$, the mean per-vector search time (top) and the maximum amount of memory used (bottom)

Table 5.1: Execution time and memory scalability.

Mean search time (ms)			L2Knng				L2Knng-a			
dataset	# rows	Δ_{sz}	$k = 25$	50	75	100	25	50	75	100
WW200-250k	250000	1.00	12.0	14.0	15.2	16.2	0.7	1.3	1.6	1.9
WW200	1017531	4.07	38.9	46.5	50.9	54.1	1.1	1.7	2.6	2.9
RCV1-100k	100000	1.00	1.0	1.3	1.5	1.7	0.1	0.2	0.2	0.3
RCV1-400k	400000	4.00	2.9	3.8	4.4	4.9	0.3	0.4	0.5	0.6
RCV1	804414	8.04	5.3	6.8	8.0	8.1	0.3	0.4	0.6	0.7

Memory usage (Gb)			L2Knng				L2Knng-a			
dataset	# rows	Δ_{sz}	$k = 25$	50	75	100	25	50	75	100
WW200-250k	250000	1.00	8.9	9.5	10.0	10.6	7.2	7.8	8.4	9.0
WW200	1017531	4.07	35.9	38.2	40.5	42.8	29.1	31.7	34.0	36.4
RCV1-100k	100000	1.00	0.9	1.1	1.4	1.6	0.7	1.0	1.2	1.4
RCV1-400k	400000	4.00	3.5	4.4	5.3	6.2	2.9	3.8	4.7	5.6
RCV1	804414	8.04	7.0	8.8	10.6	12.4	5.8	7.7	9.5	11.3

for `L2Knng` and `L2Knng-a`. The Δ_{sz} column shows the relative dataset size increase. Parameters were tuned to achieve efficient execution, and, in the case of `L2Knng-a`, high recall (95%).

The results show that the performance of both methods scales linearly compared to the dataset size. As the dataset size increases, our two methods perform better than they did for the smaller datasets (e.g., it takes much less than 8.04x the time of searching RCV1-100k to search RCV1 for 100 neighbors), while using memory directly proportional to the number of objects in the set.

Comparison with other methods

The primary goal of this work is the efficient construction of the exact k-NNG. Figure 5.9 presents execution times for the exact methods, for all six of the tested datasets. Note that execution times are log-scaled, and lower values are preferred. The *Maxscore* and *BMM* experiments on the WW200 and WW500 datasets were terminated early, after executing for 5 days, which is more than twice the execution time of `kIdxJoin` for these datasets. Additionally, Table 5.2 shows the average speedup, across all k values, of our algorithms against the best time achieved by competing approximate (left) and exact (right) methods.

`L2Knng` performed best among all exact methods, achieving over an order of magnitude improvement versus `kIdxJoin` for small values of k. The gap between our method and `kIdxJoin` is more pronounced for larger datasets than for smaller ones. Speedup

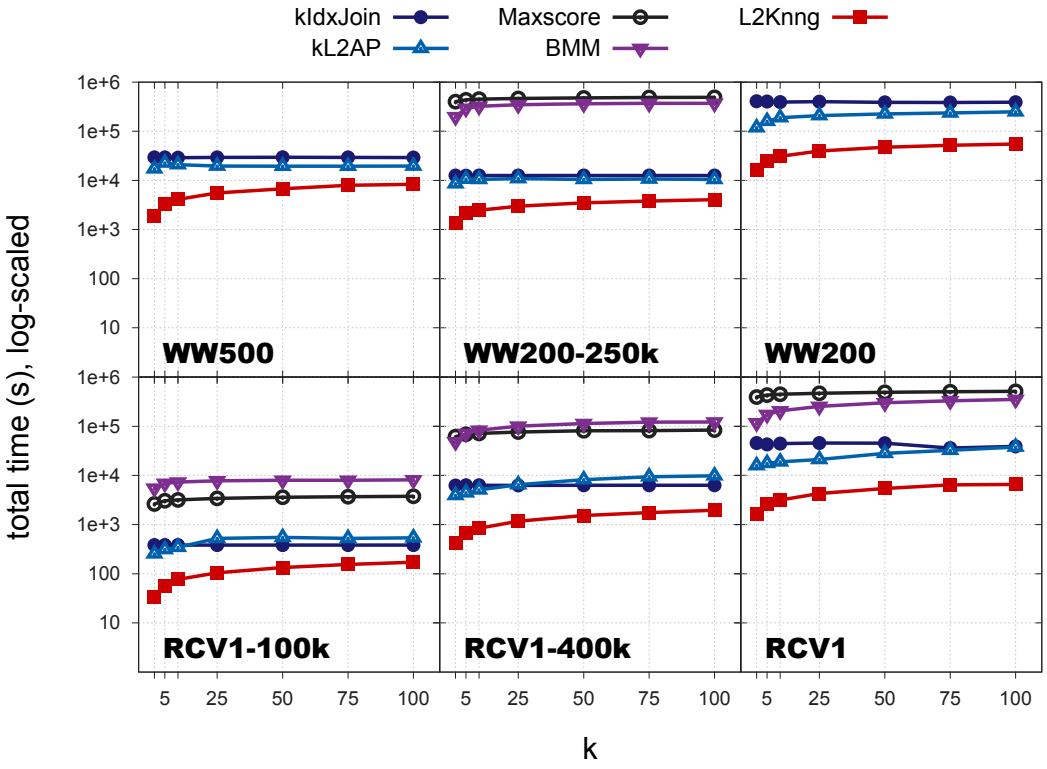

Figure 5.9: Exact k-NNG construction efficiency comparison.

of `L2Knng` vs. `kIdxJoin` ranged between 2.24–14.35x for small datasets (RCV1-100k, RCV1-400k, WW200-250k) and between 3.5–28.15x for the larger datasets. The minimum speedup for WW200, our largest tested dataset, was 7.07x.

While using a similar filtering framework as `L2Knng`, `kL2AP` performs poorly, at times taking longer to execute even than `kIdxJoin`, which is equivalent to a brute-force search. This may be due to repeated indexing in `kL2AP` and the size of its final index. As ϵ nears 0, even if we are only interested in finalizing a few neighborhoods, the inverted index lists will contain the majority of values in the dataset, and traversing it will produce many candidates. In contrast, `L2Knng` indexes each vector only once and uses *block completion* as an effective strategy to improve pruning.

Maxscore and *BMM* performed worst among all exact methods, which may be explained by the length of the query vectors used in solving the k-NNG problem. The methods were designed for short queries. They perform a sorting operation with each

Table 5.2: Average speedup of `L2Knng` and `L2Knng-a` over the best alternative.

	versus approx		versus exact	
dataset / method	`L2Knng`	`L2Knng-a`	`L2Knng`	`L2Knng-a`
WW200	0.32	9.60	5.57	178.01
WW200-250k	0.41	6.01	3.94	69.08
WW500	0.40	8.68	4.58	110.98
RCV1	0.09	0.87	6.18	61.79
RCV1-100k	0.33	2.19	4.11	27.86
RCV1-400k	0.17	1.19	5.41	40.24

Table 5.3: Execution time and scan rate for competing algorithms.

		WW500			**RCV1**		
result	method / k	10	50	100	10	50	100
time:	*Greedy Filtering*	1474.77	2885.28	4239.30	1527.78	2352.80	3809.80
	NN-Descent	60666.94	28840.24	25134.13	1961.50	2099.79	2099.78
	`L2Knng-a`	**128.82**	**389.70**	**667.30**	**286.56**	**356.52**	**596.31**
	`kIdxJoin`	29017.40	29524.21	29243.90	44465.74	45200.67	44914.38
	`kL2AP`	20948.63	19466.71	19588.10	18865.11	28028.41	37705.90
	`L2Knng`	**4104.41**	**6755.50**	**8340.02**	**3153.89**	**5439.14**	**6550.60**
scan	*Greedy Filtering*	0.0032	0.0061	0.0086	0.0031	0.0039	0.0049
rate:	*NN-Descent*	0.7614	0.9813	0.8568	0.6875	0.6890	0.6914
	`L2Knng-a`	**0.0008**	**0.0026**	**0.0045**	**0.0022**	**0.0010**	**0.0018**
	`kIdxJoin`	1.0000	1.0000	1.0000	0.8951	0.8951	0.8951
	`kL2AP`	0.4880	0.4997	0.5003	0.0060	0.0608	0.0017
	`L2Knng`	**0.0008**	**0.0025**	**0.0036**	**0.0004**	**0.0012**	**0.0013**

Best results are emphasized in bold.

query and simultaneously traverse as many inverted lists as the number of features in the query vector, which can lead to loosing cache locality. In contrast, our method traverses one inverted list at a time and updates an accumulator in increasing index order, which is much more cache friendly for long queries.

Table 5.3 presents timing and scan rate results for the three top performing exact and approximate methods. As in Section 5.2.2, we report the smallest time for which a minimum recall value of 0.95 was achieved for all approximate methods. We include results for the WW500 and RCV1 experiments, for $k \in \{10, 50, 100\}$, and use bold font to highlight the best result among approximate (top) and exact (bottom) methods, which are separated in the table by a dashed line. `L2Knng` and `L2Knng-a` achieve the lowest scan rates among the competing methods, highlighting the ability of `L2Knng-a` to

find a quality approximate solution using few similarity comparisons and the pruning ability of the `L2Knng` filtering framework. This, in turn, results in much lower execution times, both for our approximate and our exact methods, than all alternatives.

Chapter 6

Parallel Nearest Neighbor Graph Construction

In this chapter, we address multi-core parallel solutions for the exact ϵ-NNG and k-NNG construction problems using cosine similarity as a way to compare objects. The filtering framework we described in Section 4.1 is not trivial to parallelize. Awekar and Samatova [76] provide the only existing filtering based exact multi-core parallel algorithm to solve the ϵ-NNG construction problem, which we call `pAPT`. Their method is based on an existing serial algorithm they developed, `APT` [43], and uses *index sharing* as a way to allow threads to execute independent searches. As a way to better understand the intricacies involved in extracting parallelism from the framework, we will first analyze memory access patterns inherent in the computations in each stage of the framework. Then, we will present our solutions for the parallel ϵ-NNG and k-NNG construction problems.

6.1 Filtering Framework Memory Access Pattern Analysis

In this section, we analyze memory access patterns inherent in the computations in each stage of the APSS filtering framework. Additionally, we highlight the pruning choices in the `APT` algorithm by Awekar and Samatova [43] and in the `L2AP` algorithm we presented in Section 4.2, on which the parallel algorithms described in the next section are based. Table 6.1 provides a quick reference for these pruning choices.

Table 6.1: Similarity estimates in `APT`/`pAPT` and `L2AP`/`pL2AP`.

bound	stage	estimate	`APT` / `pAPT`	`L2AP` / `pL2AP`
idx	idx	$\text{sim}(\mathbf{d}_q^{\leq j}, \mathbf{d}_{>q})$	$\left\langle \mathbf{d}_q^{\leq j}, \mathbf{mx}_{\geq \mathbf{q}} \right\rangle$	$\min(\left\langle \mathbf{d}_q^{\leq j}, \mathbf{mx}_{\geq \mathbf{q}} \right\rangle, \|\mathbf{d}_q^{\leq j}\|_2)$
sz	c.g.	$\min(\|\mathbf{d}_c\|_0)$	$(\epsilon / \|\mathbf{d}_q\|_\infty)^2$	$(\epsilon / \|\mathbf{d}_q\|_\infty)^2$
rs		$\text{sim}(\mathbf{d}_q^{\leq j}, \mathbf{d}_{<q})$	$\left\langle \mathbf{d}_q^{\leq j}, \mathbf{mx} \right\rangle$	$\min(\left\langle \mathbf{d}_q^{\leq j}, \mathbf{mx} \right\rangle, \|\mathbf{d}_q^{\leq j}\|_2)$
$l2cg$		$\text{sim}(\mathbf{d}_q^{<j}, \mathbf{d}_c^{<j})$	–	$\|\mathbf{d}_q^{<j}\|_2 \|\mathbf{d}_c^{<j}\|_2$
ps	c.v.	$\text{sim}(\mathbf{d}_q, \mathbf{d}_c^{\leq})$	–	$\min(\left\langle \mathbf{d}_c^{\leq}, \mathbf{mx}_{\geq \mathbf{c}} \right\rangle, \|\mathbf{d}_c^{\leq}\|_2)$
dps_1		$\text{sim}(\mathbf{d}_q, \mathbf{d}_c^{\leq})$	$\min(\|\mathbf{d}_q\|_\infty \|\mathbf{d}_c^{\leq}\|_1, \|\mathbf{d}_q\|_1 \|\mathbf{d}_c^{\leq}\|_\infty)$	$\min(\|\mathbf{d}_q\|_0, \|\mathbf{d}_c^{\leq}\|_0) \|\mathbf{d}_q\|_\infty \|\mathbf{d}_c^{\leq}\|_\infty$
dps_2		$\text{sim}(\mathbf{d}_q, \mathbf{d}_c^{\leq})$	–	$\min(\|\mathbf{d}_q\|_0, \|\mathbf{d}_c^{\leq p}\|_0) \|\mathbf{d}_q^{\leq p}\|_\infty \|\mathbf{d}_c^{\leq p}\|_\infty$
$l2cv$		$\text{sim}(\mathbf{d}_q^{<j}, \mathbf{d}_c^{<j})$	–	$\|\mathbf{d}_q^{<j}\|_2 \|\mathbf{d}_c^{<j}\|_2$

The vectors $\mathbf{d}_q$ and $\mathbf{d}_c$ represent the query and candidate objects, respectively. Prefix and suffix vectors are defined in Section 2.1. The prefix vector $\|\mathbf{d}_c^{\leq}\|$ is the un-indexed portion of the candidate. The vector $\mathbf{mx}$ represents the max vector, containing the maximum value for each feature in the dataset. Features in the max vector $\mathbf{mx}_{\geq \mathbf{q}}$ are also upper-bounded by $\|\mathbf{d}_q\|_\infty$. The feature j represents a non-zero feature in the query and/or the candidate. The feature p is the last un-indexed candidate feature in the feature processing order that the query also has in common.

6.1.1 Indexing

Since lists in the inverted index are traversed each time a search is performed for a query object, it is beneficial to index as few values as possible. Indexing is delayed in the APSS framework until the similarity estimate of the query prefix with *any unprocessed object* reaches the threshold ϵ. Any unprocessed *neighbor*, i.e., an object with a similarity of at least ϵ with the query, is guaranteed in this way to have at least one feature in common with the query object. Then, when that neighbor is processed, the query object will be found while traversing the index.

The *partial indexing* of only suffix values in each query object improves computation efficiency by limiting the number of non-zeros traversed when identifying neighbors for a query object. In addition, it is an effective pruning strategy. Note that some objects may not have any features in common with the query suffix. These objects are automatically removed from consideration, without even starting to compare them to the query.

`APT` computes the prefix similarity estimate $\text{sim}(\mathbf{d}_q^{\leq j}, \mathbf{d}_{>q})$, which we call the *idx* bound, as the dot product between the query vector and the *max vector*, the vector made up of all maximum feature values, denoted as $\mathbf{mx}$. The estimate is improved by processing objects in decreasing order of their maximum feature weights, and bounding the max vector by the maximum feature weight in the query,

$$\text{sim}(\mathbf{d}_q^{\leq j}, \mathbf{d}_{>q})_{\texttt{APT}} = \left\langle \mathbf{d}_q^{\leq j}, \mathbf{mx}_{\geq \mathbf{q}} \right\rangle, \text{ where,}$$

$$\mathbf{mx}_{\geq \mathbf{q}} = \langle \min(mx_1, \|\mathbf{d}_q\|_\infty), \ldots, \min(mx_m, \|\mathbf{d}_q\|_\infty) \rangle.$$

In addition, `L2AP` uses the ℓ^2-norm of the query inclusive prefix ending at index j, $\|d_q^{\leq j}\|$, as an estimate of the query object similarity with any other object, which includes unprocessed objects,

$$\text{sim}(\mathbf{d}_q^{\leq j}, \mathbf{d}_{>q})_{\texttt{L2AP}} = \min(\left\langle \mathbf{d}_q^{\leq j}, \mathbf{mx}_{\geq \mathbf{q}} \right\rangle, \|\mathbf{d}_q^{\leq j}\|_2).$$

When indexing each query suffix non-zero value, `L2AP` also indexes additional meta-data, such as the ℓ^2-norm of the query prefix and its maximum value, which are used in future pruning. The similarity estimate of the un-indexed query prefix with unprocessed objects is also stored, to be used during candidate verification as an effective strategy

for pruning false positive candidates.

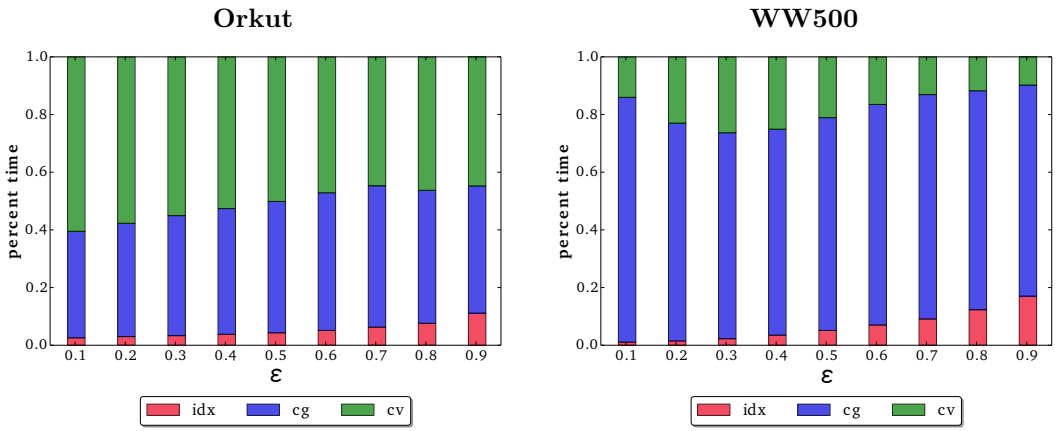

Figure 6.1: Percent execution times for the Orkut and WW500 datasets.

For each dataset, the stacked bars show the percent of search time taken by the indexing (*idx*), candidate generation (*cg*), and candidate verification (*cv*) phases in `L2AP`, for similarity thresholds ranging from 0.1 to 0.9.

Indexing requires traversing the sparse query vector and accessing values in the max vector, which are both stored in memory as arrays. A sparse vector is stored in memory in two arrays, one containing feature IDs and the other their associated values. Since indexing occurs only once for each object in the set, it takes much less of the overall search time than the other two stages in the framework. As an example, Figure 6.1 shows the percent of overall search time taken by each of the three stages in `L2AP`, for ϵ ranging from 0.1 to 0.9, for a network (Orkut) and a text-based dataset (WW500). Furthermore, values in both the query vector and feature maximum values are accessed sequentially, in sorted feature processing order, and can take advantage of software and hardware pre-fetching to reduce latency. As a result, we will focus on optimizing the other two stages in the framework. It is important to note, however, that the size of the inverted index is highly dependent on the similarity threshold ϵ. As shown in Figure 4.2, higher thresholds allow delaying indexing further and lead to a smaller inverted index, and thus more potential candidates being automatically pruned.

6.1.2 Candidate Generation

During the candidate generation stage of the framework, the lists in the current version of the inverted index associated with non-zero feature values in the query object are scanned, one list at a time. An accumulator is used to keep track of partial dot-products between the query and encountered objects. Once accumulation has started for an object, it becomes a *candidate*.

Accumulation is prevented for a new object in two ways. First, the size of the candidate vector (number of non-zeros) is checked against a minimum size estimate, which we call the *size* (sz) bound, and candidates with too few non-zeros are ignored. Both `APT` and `L2AP`[1] use the same bound in this step. Second, no *new candidates* are accepted if the query prefix does not have enough weight to achieve at least ϵ similarity with an indexed object. Index lists are traversed in inverse feature processing order, and the similarity of the query prefix with any indexed object, $\text{sim}(\mathbf{d}_q^{\leq j}, \mathbf{d}_{<q})$, which we call the *remaining similarity* (rs) bound, is used to decide whether to accept new candidates. In `APT`, the approximation is based on computing the similarity of the query with the max vector, while `L2AP` additionally bounds it by the prefix ℓ^2-norm of the query,

$$\text{sim}(\mathbf{d}_q^{\leq j}, \mathbf{d}_{<q})_{\texttt{APT}} = \left\langle \mathbf{d}_q^{\leq j}, \mathbf{mx} \right\rangle,$$
$$\text{sim}(\mathbf{d}_q^{\leq j}, \mathbf{d}_{<q})_{\texttt{L2AP}} = \min(\left\langle \mathbf{d}_q^{\leq j}, \mathbf{mx} \right\rangle, \|\mathbf{d}_q^{\leq j}\|_2).$$

While accumulating partial dot-products with candidates, at each feature that both the query and candidate have in common, `L2AP` also checks an additional bound, named $l2cg$. The $l2cg$ bound is based on estimating the prefix similarity up to that feature, leveraging the Cauchy-Schwarz inequality, as

$$\text{sim}(\mathbf{d}_q^{<j}, \mathbf{d}_c^{<j}) = \|\mathbf{d}_q^{<j}\|_2 \|\mathbf{d}_c^{<j}\|_2.$$

The critical memory access portions of the candidate generation stage are updating values in the accumulator data structure, which can be reused for each query, and

[1] Note that [18] uses a different sz bound, $\epsilon/(\|\mathbf{d}_q\|_\infty \|\mathbf{d}_c\|_\infty)$, and erroneously states it is superior to $(\epsilon/\|\mathbf{d}_q\|_\infty)^2$. We found both bounds provide limited benefit for different values of ϵ, and chose to use the same bound as `APT` here to simplify comparison.

traversing index lists. If these structures take up more than the available cache memory, the computation will be delayed while data is loaded from main memory.

Due to the predefined object processing order, objects that do not meet the minimum size requirement when traversing the index will also not meet the requirement for future query objects and can be removed from the index. Removing objects from the index is a costly operation, and `APT` instead updates inverted list start pointers, effectively removing objects from the start of the list until an object of adequate size is found. These objects will not need to be traversed in future iterations and can speed up computation. Experiments detailed in Section 4.4 showed this technique had limited benefit and `L2AP` does not use it.

6.1.3 Candidate Verification

Candidate verification iterates through the list of candidates and computes the partial similarity between the query vector and the un-indexed portion of each candidate, adding it to the already accumulated similarity. Each candidate is first vetted based on an upper bound of its un-indexed prefix similarity with any object stored during indexing. `APT` uses the Hölder inequality to derive this bound, which we name dps_1, as

$$\text{sim}(\mathbf{d}_q, \mathbf{d}_c^{\leq})_{\texttt{APT}} = \min(\|\mathbf{d}_q\|_\infty \|\mathbf{d}_c^{\leq}\|_1, \|\mathbf{d}_q\|_1 \|\mathbf{d}_c^{\leq}\|_\infty).$$

`L2AP` uses several different estimate here. First, since the query follows the candidate in processing order, the similarity $\text{sim}(\mathbf{d}_q, \mathbf{d}_c^{\leq})$ can be approximated as the similarity $\text{sim}(\mathbf{d}_c^{\leq}, \mathbf{d}_{>c})$, which was computed and stored while indexing $\mathbf{d}_c$, and is equivalent to

$$\text{sim}(\mathbf{d}_q, \mathbf{d}_c^{\leq})_{\texttt{L2AP}} = \min(\langle \mathbf{d}_c^{\leq}, \mathbf{mx}_{\geq \mathbf{c}} \rangle, \|\mathbf{d}_c^{\leq}\|_2).$$

We call this bound ps. Second, `L2AP` uses a different dps_1 bound that, while theoretically inferior to the one in `APT` with regards to candidate pruning, was slightly more efficient in experiments on a wide range of datasets (detailed in Section 4.4),

$$\text{sim}(\mathbf{d}_q, \mathbf{d}_c^{\leq})_{\texttt{L2AP}} = \min(\|\mathbf{d}_q\|_0, \|\mathbf{d}_c^{\leq}\|_0) \|\mathbf{d}_q\|_\infty \|\mathbf{d}_c^{\leq}\|_\infty.$$

Third, after finding the last un-indexed candidate feature p in the feature processing order that is also present in the query, `L2AP` checks a tighter version of the dps_1, bound, which we call dps_2,

$$\text{sim}(\mathbf{d}_q, \mathbf{d}_c^{\leq})_{\texttt{L2AP}} = \min(\|\mathbf{d}_q\|_0, \|\mathbf{d}_c^{\leq p}\|_0)\|\mathbf{d}_q^{\leq p}\|_\infty \|\mathbf{d}_c^{\leq p}\|_\infty.$$

Finally, while computing the prefix dot-product, at each common feature, `L2AP` first checks the Cauchy-Schwarz inequality based estimate, which here we call $l2cv$,

$$\text{sim}(\mathbf{d}_q^{<j}, \mathbf{d}_c^{<j}) = \|\mathbf{d}_q^{<j}\|_2 \|\mathbf{d}_c^{<j}\|_2.$$

The accumulator is not critical in the candidate verification stage, as processing occurs for one candidate at a time. The partial accumulated similarity of a candidate can be looked up once and stored in a local variable. On the other hand, feature values and meta-data associated with those features in the query vector are accessed in a random fashion, based on the features encountered in the candidate object. To facilitate computing dot products between the query and candidate vectors, we have found it beneficial to insert the feature values of the query vector, its prefix ℓ^2-norm values, and its prefix maximum values in a hash table. When iterating through the sparse version of a candidate object's un-indexed prefix, the query feature, prefix maximum and ℓ^2-norm values can then be quickly looked up in $O(1)$ time. The cost of using a hash table can be offset by reusing the structure for verifying many candidates. An alternative to looking up query values in a hash table would be to traverse the candidate and query vectors concurrently, assuming a predefined global feature traversal order. We have found that in most cases (other than datasets with small number of vector non-zeros) this strategy leads to 2x-3x slower execution times.

6.2 Parallel Cosine ϵ-NNG Construction

In this section, we present two parallel solutions to the APSS problem. First, we summarize algorithmic choices in the method of Awekar and Samatova, `pAPT`. We then introduce `pL2AP`, which was designed based on the memory access observations we made in Section 6.1, with the goal of improving cache locality during similarity search.

6.2.1 pAPT

Awekar and Samatova introduced the first multi-core parallel APSS algorithm [76], `pAPT`, based on their serial `APT` algorithm, which we describe in Algorithm 11. Their main idea was to pre-compute the partial inverted index (lines 4–5), rather than indexing each object after its processing, and allow threads to share the index structure. To prevent synchronization overheads when removing values associated with short vectors from the inverted index, `pAPT` duplicates, for each thread, a list of offsets from the beginning of each inverted list. Then, each thread modifies its own offsets, incrementing them to remove only items at the start of inverted lists.

Algorithm 11 The `pAPT` algorithm.

1: **function** pAPT(D, ϵ)
2: Set processing order for vectors and/or features
3: $O \leftarrow \varnothing,\ I_j \leftarrow \varnothing,$ **for** $j = 1, \ldots, m$
4: **for each** $q = 1, \ldots, n$ **do**
5: Index($\mathbf{d}_q, \mathcal{I}, \epsilon$)
6: **for each** $q = 1, \ldots, n$, **in parallel do**
7: $\mathbf{c}_q \leftarrow$ GenerateCandidates($\mathbf{d}_q, \mathcal{I}, \epsilon$)
8: $O \leftarrow O \cup$ VerifyCandidates($\mathbf{d}_q, \mathbf{c}_q, \mathcal{I}, \epsilon$)
9: **return** O

Awekar and Samatova proposed three load balancing strategies in `pAPT`: block, round-robin, and dynamic partitioning. The object processing order in the filtering framework, namely in decreasing maximum value order, after first normalizing object vectors, means that objects with few non-zeros are processed first, and those with many non-zeros last. As a result, statically assigning n/nt consecutive objects to each thread, where nt is the number of threads, leads to load imbalance. Awekar and Samatova attempted to fix the potential imbalance by assigning subsets of query objects with equal number of non-zeros to each thread, but found this strategy is still worse than round-robin or dynamic partitioning. The best performing load balancing strategy in their experiments was dynamic partitioning, which assigns a small set of objects to a thread as soon as it has finished processing its previous assigned set.

6.2.2 pL2AP

Our new method, `pL2AP`, uses the same indexing, candidate generation and verification pruning choices as `L2AP`. Similar to `pAPT`, it indexes all objects first, and allows threads to share the index structure during the search. Additionally, `pL2AP` employs two strategies aimed at improving cache locality during search. First, cache-tiling breaks up the inverted index into blocks that can fit in the system cache, reducing latency during candidate generation. Second, for datasets with high dimensionality, mask-based hash tables can greatly reduce the amount of memory required for storing query object values and meta-data during search, allowing them to persist in the cache during candidate verification. Algorithm 12 provides an overview of our method.

Algorithm 12 The `pL2AP` algorithm.

1: **function** pL2AP($D, \epsilon, h, \zeta, \eta$)
2: Set processing order for vectors and features
3: **for each** $q = 1, \ldots, n$ **in parallel do**
4: $S \leftarrow \text{FindIndexSplit}(\mathbf{d}_q, \epsilon)$
5: $K \leftarrow \text{FindIndexAssignments}(S, \zeta)$
6: $O \leftarrow \varnothing,\ I_{k,j} \leftarrow \varnothing,$ **for** $j = 1, \ldots, m$ **and** $k = 1, \ldots, K$
7: **for each** $q = 1, \ldots, n$ **do**
8: $\text{Index}(\mathbf{d}_q, \mathcal{I}, S, \epsilon)$
9: **for each** $k = 1, \ldots, K$ **do**
10: **for each** $l = S[k], \ldots, n$, **in increments of** η **do**
11: **for each** $q = l, \ldots, \min(l+\eta\ -1,\ \text{n})$, **in parallel do**
12: $\mathbf{c}_q \leftarrow \text{GenerateCandidates}(\mathbf{d}_q, \mathcal{I}_k, \epsilon)$
13: $O \leftarrow O \cup \text{VerifyCandidates}(\mathbf{d}_q, \mathbf{c}_q, \mathcal{I}_k, \epsilon)$
14: **return** O

Cache-tiling

Cache-tiling is designed to increase cache locality during the candidate generation stage of the similarity search by ensuring the inverted index and accumulator structures fit in cache. To achieve this, the inverted index is split into several consecutive sections, called *tiles*, and each index is used in turn to find neighbors. Choosing the size of each cache tile is non-trivial in the APSS problem, due to the varying number of feature values being indexed for each object. For example, choosing to index the same number of objects in each tile will lead to large indexes for the final tiles to be processed, which

may not fit in cache. Instead, `pL2AP` first finds the first feature to be indexed in each object (line 4), which also provides the number of values to be indexed in each object. These counts are used to define the consecutive sets of objects to be indexed together in each tile. The list S, containing tile start and end offsets given the predefined processing order, is then used to index each object suffix in their assigned inverted index (line 8).

We use an array to track accumulated similarities for candidates. Since the accumulation array is randomly accessed for different candidates encountered while traversing the inverted index, nt accumulation arrays should also fit in cache along with the index, one for each thread. The size of the accumulation array is the same as the number of objects assigned to an index.

The un-indexed portion of each un-pruned candidate vector is sequentially accessed during candidate verification. To maximize cache locality, we explicitly create a sparse *forward index* containing only the prefix values for objects in each tile.

During parallel sections (lines 3 and 11), `pL2AP` follows a dynamic task partitioning approach, assigning a small set of objects to a thread to process as soon as it has finished processing its previous assigned set. Since candidate pruning is unpredictable, a thread may get assigned objects that finish processing quickly and may jump ahead many places in the processing order. This may lead to loss of cache locality if some threads read query objects from different portions of the dataset. To prevent this, we process queries η at a time, in a block synchronous fashion, where η is an input parameter, forcing threads to read from the same subset of query vectors, which should be located in close proximity in memory.

Query vector mask-hashing

During candidate verification, `pL2AP` traverses the candidate prefix, and checks whether the query has non-zero values for the encountered features. When a common feature is found, query object meta-data (prefix ℓ^2-norm or maximum value) are used to check whether the candidate can be pruned. An efficient way to locate query vector values and meta-data during this process is to store them in arrays, as dense vectors. However, for datasets with high dimensionality (generally above 10^6), this technique can lead to polluting the cache with zero values from the dense arrays, evicting other necessary data.

Given that query vectors are sparse, and their features are always processed in a predefined order, we developed a heuristic hash-table data structure that uses a small amount of cache space, takes advantage of O(1) access times for most look-ups and leads to few collisions in practice. A small array of size $h + \max(\|d_q\|_0) - 1$ is used in `pL2AP` to store matching offsets in one or more lists containing the query data. Here, $h = 2^\alpha$ ($\alpha \geq 0$) is a predefined parameter, generally much smaller than m, and $\max(\|d_q\|_0)$ is the maximum number of non-zero features for any object. An efficient hashing function maps feature IDs to the $[0, h-1]$ domain, and collisions are entered in the hash-table array in order, in an *overflow* section starting with index h. Since partial dot-product computations with candidates follow the same traversal order, collisions can be quickly resolved by traversing only a subset of the overflow features. In practice, however, we have found that less than 1% of hash key look-ups end in collision.

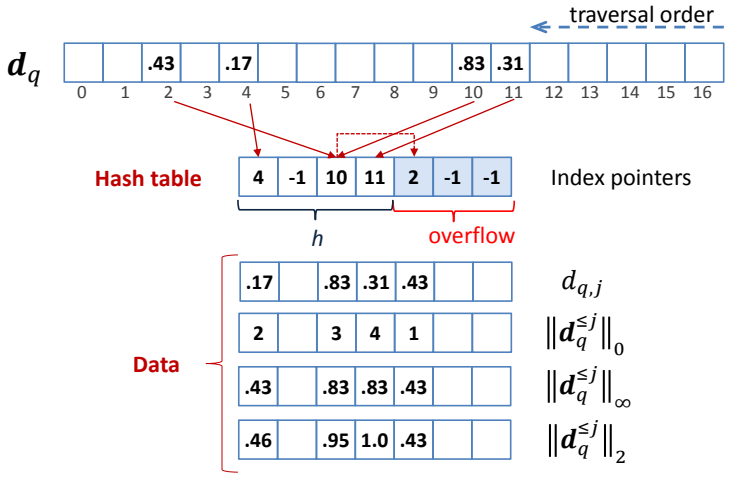

Figure 6.2: Example query hash table use in `pL2AP`.

Figure 6.2 provides an example of how a query object might use the hash table in `pL2AP`, for $h = 2^2$. The hash table array is initialized with negative values. Traversing the query non-zeros in reverse feature processing order, the 11th query feature is mapped to the 4th hash table cell, via an efficient truncate operation, $11 \;\&\; (4-1)$, where & is the bitwise AND logical operator. The feature ID is stored in the hash table at the mapped key index, and one or more value arrays are populated with salient information about the query at the same key index location. `PL2AP` tracks the query prefix value, size,

maximum value, and ℓ^2-norm at each index, which are used to check different pruning bounds. In a similar fashion, the 10th query feature is mapped to the 3rd hash table cell, and the 4th query feature to the 1st hash table cell. When mapping the 2nd query feature, the collision is handled by entering the item in the overflow part of the hash table array, in traversal order. When verifying a candidate d_c, its forward index features are traversed in the same order as the query was traversed. Thus, when collisions occur, they can be found by partially traversing the overflow section of the hash table, keeping a pointer to the last cell with a feature ID greater or equal than the sought ID.

To avoid excessive collisions, `pL2AP` dynamically chooses whether to use the hash-table or dense arrays for the query object data. Specifically, objects with less than $h/2^3$ non-zeros will use the hash-table data structure, while the rest will use dense vector representations of the query and meta-data vectors.

6.3 Experimental Evaluation for Parallel Cosine ϵ-NNG Construction

In this section, we present our experimental results for parallel cosine ϵ-NNG construction, along two directions. First, we analyze pruning effectiveness, cache locality improvement, and parameter sensitivity in `pL2AP`. We compare pruning effectiveness of `pAPT` and `pL2AP` and find `pL2AP` is a lot more effective at pruning objects than `pAPT`, especially for text datasets. Second, we report the execution efficiency of `pL2AP`, comparing it with several serial and parallel baselines, analyze the scaling characteristics of parallel methods, and measure the amount of load imbalance in the `pL2AP` execution.

6.3.1 Baseline Approaches

In addition to the `pAPT` algorithm by Awekar and Samatova, which we described in Section 6.2, we compare `pL2AP` against the following algorithms.

1. `IdxJoin`, `APT`, and `L2AP` are baseline serial APSS search methods described in detail in Sections 4.2–4.4. We report speedup over the fastest execution time of any of the serial methods.
2. `pIdxJoin` uses similar cache-tiling as `pL2AP`, but does not use any pruning when

computing similarities. For each block of queries, `pIdxJoin` sequentially retrieves a block of objects to search against and indexes all their values. Threads then share the index to compute similarities, via accumulation, of each assigned query object against all indexed objects, retaining those resulting pairs above the threshold ϵ.

3. $\texttt{pL2AP}_{\texttt{rr}}$ follows the same parallelism strategy as `pAPT` (see Section 6.2), but takes advantage of the advanced pruning bounds of `L2AP`. After first indexing the suffixes of all objects, $\texttt{pL2AP}_{\texttt{rr}}$ dynamically assigns small sets of query objects for processing to available threads. For each query object, $\texttt{pL2AP}_{\texttt{rr}}$ indexes the same values and performs the same pruning in the candidate generation and verification stages as `pL2AP`.

6.3.2 Pruning Effectiveness

Pruning effectiveness comparison with pAPT

Both our method, `pL2AP`, and the shared memory parallel baseline `pAPT`, follow the same strategy in solving the APSS problem. They build a partial inverted index that is used to identify, for each query object, a list of candidates the query should be compared with. While comparing query objects with candidates, they prune as many un-promising pairs as possible, and in the end fully compute the dot-product of a small subset of the candidate list, which is a superset of the nearest neighbors. While their serial computation strategy is the same, the two methods rely on different theoretic similarity upper bounds to decide which values in the query object should be indexed, whether an object should become a candidate, and when a candidate should be pruned.

Indexing fewer values can speed up index traversal and thus lead to performance improvements. In addition, it will lead to shorter candidate lists being generated. Considering fewer candidates, as well as more aggressive pruning, can lead to fewer dot-products being computed in full and to better performance. Figure 6.3 shows the number of indexed non-zeros, candidates, and dot-products when executing `pL2AP`, normalized by the respective values when executing `pAPT`, for ϵ between 0.3 and 0.9, for all six datasets. As compared to `pAPT`, our method generally indexes fewer values, considers fewer candidates, and evaluates fewer complete dot-products, especially at high similarity values. While the difference in the number of indexed values and candidates

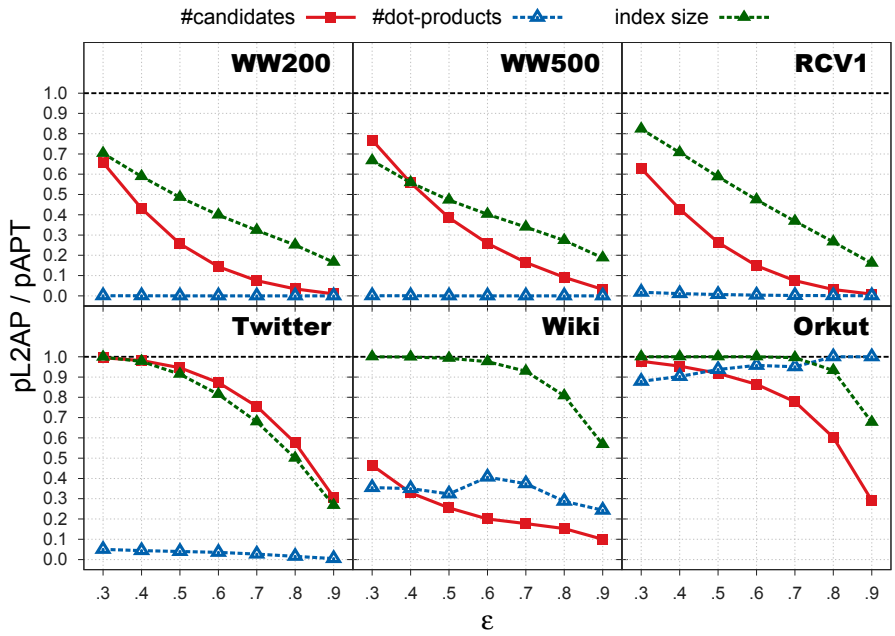

Figure 6.3: Index size, number of candidates, and number of dot-products in `pL2AP` executions vs. respective values in `pAPT`.

is smaller at $\epsilon = 0.3$, `pL2AP` is able to prune a much higher number of candidates than `pAPT` in all datasets except Orkut, highlighting the improved pruning effectiveness in our method. Orkut is a binary dataset, with short columns that deviate little in length, as shown in Section 2.3. After our pre-processing, objects in the Orkut set will have fairly uniform small values, which contribute little to accumulation operations and cannot be approximated well. While the size of the un-pruned set of candidates in `pL2AP` was in most cases between 1–3x the size of the set of true neighbors, it ranged between 13–137x for the Orkut dataset. The `pAPT` method had a similar high number of un-pruned candidates for Orkut, highlighting the inherent similarity estimation difficulty for this dataset.

Pruning effectiveness in pL2AP

Our method works by pruning the majority of the candidates that are not true neighbors. Once an object becomes a candidate, it can be pruned by the $l2cg$ bound while accumulating values traversing the inverted index in the candidate generation stage (cg

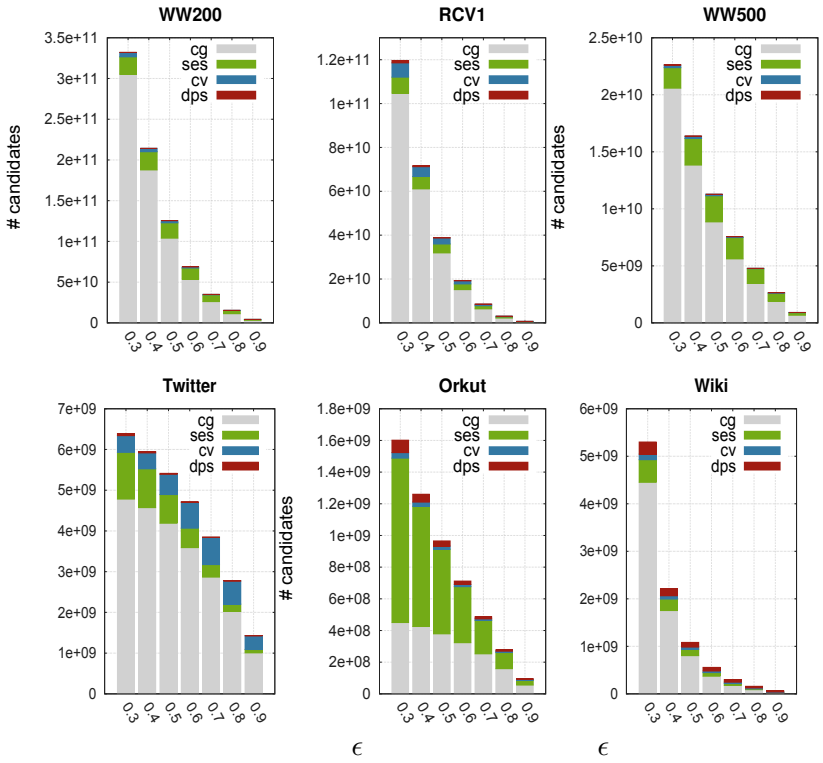

Figure 6.4: Candidate pruning in pL2AP.

in figures), when checking the ps, dps_1 and dps_2 prefix similarity estimate bounds at the onset of the candidate verification stage (*ses* in figures), or by the $l2cv$ bound while accumulating values traversing the forward index in the candidate verification stage (*cv* in figures). Earlier pruning of candidates means less time spent accumulating dot-products in vain and will lead to improved performance. In an experiment in which we used consistent parameters for all datasets ($h = 2^{13}$, $\eta = 25K$, and $\zeta = 1M$), we counted the number of candidates pruned in each stage of the algorithm. We report these values in Figure 6.4, for all datasets and ϵ values, along with the number of candidates that were not pruned and had their dot-products computed in full (*dps* in figures).

Results show that pL2AP prunes the majority of objects soon after they become candidates, in the candidate generation stage (*cg*). Most of the remaining objects are pruned by the *ses* bound, which is checked once, at the beginning of the candidate verification stage, and by additional pruning in the candidate verification stage (*cv*).

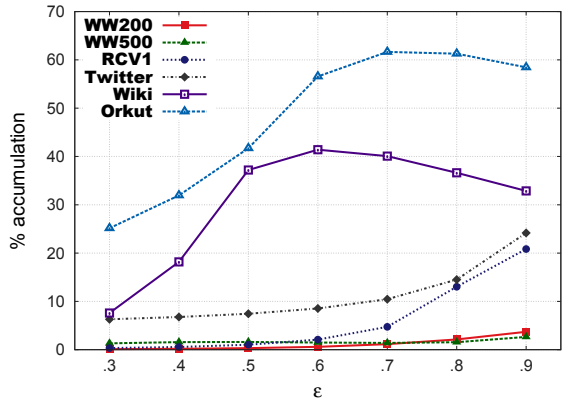

Figure 6.5: Mean percent accumulated non-zeros before pruning in `pL2AP`.

At $\epsilon = 0.3$, for example, only 0.02%–4.89% of candidates survived all pruning across our datasets.

A large number of objects never become candidates in `pL2AP`, as a result of either the ℓ^2-norm based candidate acceptance bound in the candidate generation stage of the algorithm, or due to the prefix-filtering based index reduction. On average, across all ϵ values, only 0.7%–50.4% of all potential candidates actually became candidates for our datasets. Of those, most are pruned quickly, in the first stage of our method. As a way to gauge how quickly candidates are pruned, we measured the number of executed multiply-adds versus the number of possible multiply-adds (percent of accumulated non-zeros) in the similarity computation of each pruned candidate. In Figure 6.5, we report the mean percent accumulated non-zeros for our six datasets. In each experiment, we used consistent parameters for all datasets (1 thread, $h = 2^{13}$, $\eta = 25K$, and $\zeta = 1M$).

The results show that, for most of the datasets and ϵ values, `pL2AP` accumulates much less than 10% of the common non-zeros between a query and a non-neighbor candidate on average. Before pruning unsuitable candidates, `pL2AP` generally accumulates less than 4% of the common non-zeros for text datasets, but it traverses between 10%–60% of the common values for network datasets with short rows, like Wiki and Orkut.

Cache locality improvements in pL2AP

While `pL2AP` performs the same pruning as `L2AP`, it scans each query object multiple times to compare against objects in multiple constructed inverted indexes. The smaller

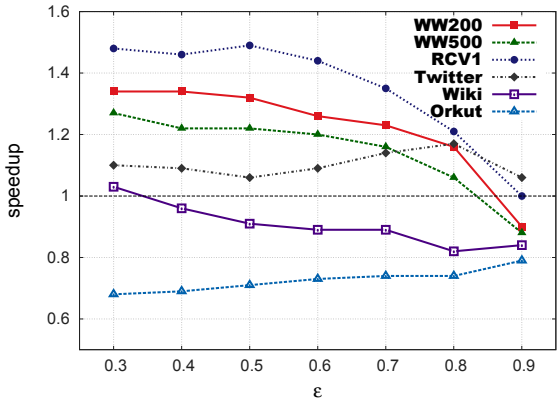

Figure 6.6: Speedup of 1-threaded `pL2AP` over `L2AP`.

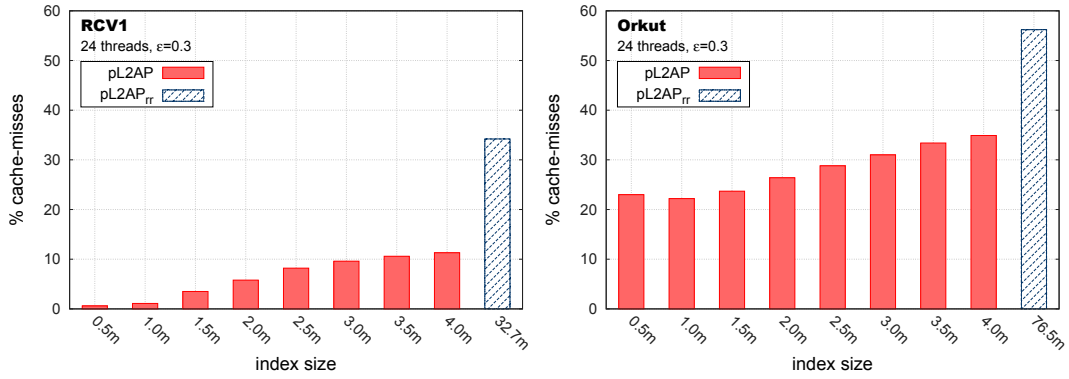

Figure 6.7: Percent cache misses of $\texttt{pL2AP}_{\texttt{rr}}$ and `pL2AP` with ζ between 1.5M and 4M non-zeros for the RCV1 (top) and Orkut (bottom) datasets.

inverted indexes and the mask-based hash table used during the search help avoid cache thrashing, improving efficiency by reducing time wasted waiting for data transfers from memory to cache. To measure the serial effect of this improvement, we compared the 1-threaded execution of `pL2AP` against the serial `L2AP` algorithm. We used $\eta = 25K$ objects and $\zeta = 1M$ non-zeros for this test. Figure 6.6 shows speedup results for each of the six datasets we tested, for ϵ between 0.3 and 0.9. The results show an improvement over `L2AP` for datasets with long inverted lists, whether text or network based. The short inverted lists in the Orkut and Wiki dataset do not provide enough cache reuse for 1 thread to hide the additional work of multiple query searches, leading to slower execution than that of `L2AP`.

The small inverted index in pL2AP is shared by all threads in executing concurrent searches. As another way to quantify cache locality improvements, we compared the percent of cache misses when executing pL2AP and $\texttt{pL2AP}_{\texttt{rr}}$ with 24 threads. Both algorithms perform the same pruning, but $\texttt{pL2AP}_{\texttt{rr}}$ builds a single inverted index and does not consider cache locality in its execution. We used the *perf* Linux utility to count the number of cache references and cache misses. Figure 6.7 shows our results when executing pL2AP with ζ between $0.5M$ and $4M$ non-zeros and $\texttt{pL2AP}_{\texttt{rr}}$, on the RCV1 (left) and Orkut (right) datasets, for $\epsilon = 0.3$. We show the size of the inverted index that $\texttt{pL2AP}_{\texttt{rr}}$ builds below its bar in the graph. We observed similar results for most other datasets and ϵ values. In general, pL2AP improves cache locality, and the improvement is more pronounced for text based datasets, which tend to have longer inverted lists. The percent cache misses for RCV1, for example, was reduced from 35% to less than 2% at $\zeta = 0.5$M.

Parameter sensitivity

Our method, pL2AP, is controlled by three parameters. The size of the mask-based hash table, h, is dependent on the dimensionality of the feature space. Choosing a small h value for a dataset with large dimensionality will likely cause many hash table collisions and slow down execution. Similarly, the ζ parameter dictates the number of non-zeros that should be included in each inverted index, which dynamically decides the size of each cache tile. Choosing a small ζ value will lead to many inverted indexes being created which may lead to slow-downs due to repeated traversals of the query objects. On the other hand, choosing an ζ value that is too large will diminish the cache locality benefits of our tiling strategy. To ascertain the sensitivity of pL2AP to these parameter choices, we tested different values of each parameter while keeping the other two unchanged.

In the first experiment, we set ζ to $1M$ non-zeros and η to $25K$ and varied h between 2^5 and 2^{15}. Results of these experiments over our six datasets are shown on the left side of Figure 6.8 (figure is best viewed in color), as execution times relative to the $h = 2^{13}$ parameter choice for each dataset. Our method is not sensitive to this parameter for text and the Twitter datasets, which have smaller dimensionality, but can incur over 2.5x slowdown when choosing a small hash table size for the Orkut or Wiki datasets,

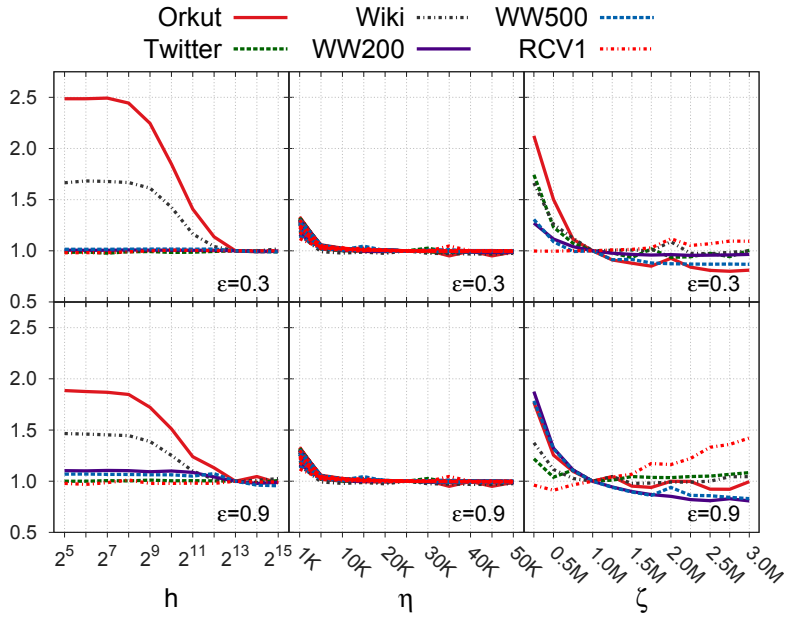

Figure 6.8: Relative execution times for different h, η, and ζ parameter choices.

which both have over $3M$ dimensions.

The middle section of Figure 6.8 shows execution times for each dataset, given $h = 2^{13}$ and $\zeta = 1M$, for η between $1K$ and $50K$, relative to the execution time for $\eta = 25K$. We found that choosing the size of each bulk synchronous block, η, does not affect performance in `pL2AP`, as long as the η value is not too small. We found any values above $5K$ to be adequate for all datasets.

Finally, we tested the sensitivity of the ζ parameter, for values between $0.25M$ and $3.0M$, given $\eta = 25K$ and $h = 2^{13}$, and show times relative to the $\zeta = 1M$ execution in the right section of Figure 6.8. While the ζ choice will be dependent on the cache configuration of the target system, our experiments showed that `pL2AP` performed well for most datasets given ζ set to at least $1M$ non-zeros.

Table 6.2: Tested `pL2AP` pruning strategies.

Strategy	Bounds checked	Index update
base	$\{idx, rs, ps, l2cg, l2cv, dps_1\}$	no
sz	base + $\{sz\}$	no
dp	base + $\{dps_2\}$	no
szdp	base + $\{sz, dps_2\}$	no
szdpupd	base + $\{sz, dps_2\}$	yes

6.3.3 Execution Efficiency

Pruning choices in pL2AP

Pruning is an effective mechanism for reducing the number of similarity computations that must be executed to solve the APSS problem. However, bounds checking incurs additional costs which may not outweigh their benefit. Previous experiments presented in Section 4.4 proved the effectiveness of our ℓ^2-norm based bounds in each stage of the search framework, and showed the sz and dps_2 bounds had little effect in general over the search efficiency. As a way to quantify this effect when executing with multiple concurrent threads, we tested `pL2AP` in four configuration scenarios, listed in Table 6.2. The "base" configuration did not effect any pruning based on the sz or dps_2 bounds. The "sz" and "dp" configurations enabled pruning based on the sz and dps_2 bounds, respectively, and the "szdp" configuration enabled pruning based on both the sz and dps_2 bounds. When checking the sz bound, `pAPT` removes values associated with short vectors from the beginning of inverted lists, which can potentially improve efficiency. We added this capability to `pL2AP` and tested it in the configuration "szdpupd", which enables all pruning strategies and also performs index updates. Using the same input parameters for all datasets ($nt = 24$, $h = 2^{13}$, $\eta = 25K$ and $\zeta = 1M$), we recorded search execution times under each scenario.

Table 6.3 reports the results of our experiment. For each ϵ value, times in all configuration scenarios were normalized by that of the sz scenario, and we report the mean, standard deviation (std), minimum and maximum of experiment results across all ϵ values. The best performing results are highlighted in bold. The sz and dp configurations showed little improvement over the base one, at times leading to slower execution times. Checking the sz bound was beneficial in most cases, especially for network datasets, and had better performance than checking the dps_2 bound instead.

Table 6.3: Performance of different pruning choice configurations in `pL2AP`.

versus	mean	stdv	min	max	mean	stdv	min	max
	Orkut				WW200			
nbase	1.0642	0.0237	1.0234	1.0929	**0.9933**	0.0163	0.9719	1.0240
dp6	1.1017	0.0249	1.0684	1.1377	1.0034	0.0112	0.9948	1.0296
sz	**1.0000**	0.0000	1.0000	1.0000	1.0000	0.0000	1.0000	1.0000
szdp	1.0443	0.0082	1.0319	1.0606	1.0152	0.0119	1.0047	1.0369
szdpupd	1.4624	0.0714	1.3222	1.5298	1.0661	0.0481	1.0139	1.1586
	Twitter				WW500			
nbase	1.0191	0.0127	0.9974	1.0345	**0.9980**	0.0109	0.9894	1.0234
dp6	1.0416	0.0230	0.9869	1.0581	1.0115	0.0178	0.9975	1.0540
sz	**1.0000**	0.0000	1.0000	1.0000	1.0000	0.0000	1.0000	1.0000
szdp	1.0416	0.0219	1.0156	1.0736	1.0249	0.0212	1.0097	1.0744
szdpupd	1.0671	0.0273	1.0345	1.1279	1.0442	0.0354	1.0145	1.1241
	Wiki				RCV1			
nbase	1.0373	0.0109	1.0190	1.0540	1.0017	0.0051	0.9909	1.0086
dp6	1.0540	0.0118	1.0305	1.0675	1.0090	0.0061	1.0027	1.0215
sz	**1.0000**	0.0000	1.0000	1.0000	**1.0000**	0.0000	1.0000	1.0000
szdp	1.0128	0.0051	1.0076	1.0243	1.0104	0.0055	1.0029	1.0212
szdpupd	1.2326	0.0489	1.1484	1.3092	1.0240	0.0137	1.0108	1.0515

Execution times for each configuration were normalized by respective execution times of the *sz* configuration. We present the mean, standard deviation (stdv), minimum and maximum of experiment results across all ϵ values, given $h = 2^{13}$, $\eta = 25K$ and $\zeta = 1M$ input parameters. The best mean performance is highlighted with bold.

The combined scenario szdp did not perform better than the sz scenario on average. The results in the remainder of this work assume the sz configuration.

In general, the index update strategy did not improve performance. For network datasets with many short inverted lists, its execution was 1.22–1.40x slower than that of the szdp configuration, which effected the same pruning without updating the index. The worse efficiency is likely due to loss of cache locality having to interrupt traversing inverted lists to update their start pointer, as well as copying the list of pointers for each thread, which in `pL2AP` occurs for each constructed inverted index.

Comparison with serial methods

We compared the execution time of all parallel methods, executed with 24 threads, with the best serial execution time achieved by any of the serial algorithms. Figure 6.9

shows the results of this experiment. In all cases, pL2AP had the best execution time of all parallel methods, achieving speedups of 2–20x for network datasets and 12–34x for text datasets. Compared to existing parallel baselines, pL2AP executed 1.5–3x faster for network datasets and 7–238x faster for text datasets. While $\texttt{pL2AP}_{\texttt{rr}}$ uses the same type of pruning as pL2AP, it traverses the entire inverted index during each query and, as a result, cannot perform as well. Instead, by using tiling and other optimizations that promote cache locality, pL2AP is able to achieve very good speedup for datasets with long inverted index lists, such as text datasets. At high similarity thresholds, however, pL2AP is able to prune candidates quickly and does not need to traverse many candidate and query vector features, rendering our cache locality optimizations less effective.

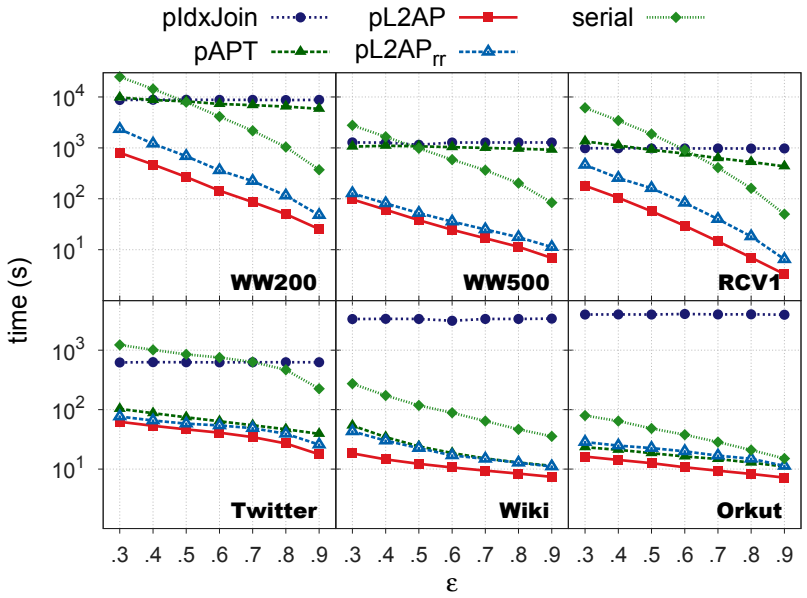

Figure 6.9: Execution times of parallel methods and the best serial alternative.

As expected, the pIdxJoin algorithm, which does not perform any pruning, was very slow in comparison to the other parallel methods. It performed very poorly on network datasets, much slower even than L2AP, the fastest serial method, potentially due to their high dimensionality. The pAPT method of Awekar and Samatova performed fairly well on network datasets, but was very slow on text datasets. It was not able to prune as many candidates as pL2AP in general, and ended up performing many more unnecessary similarity computations.

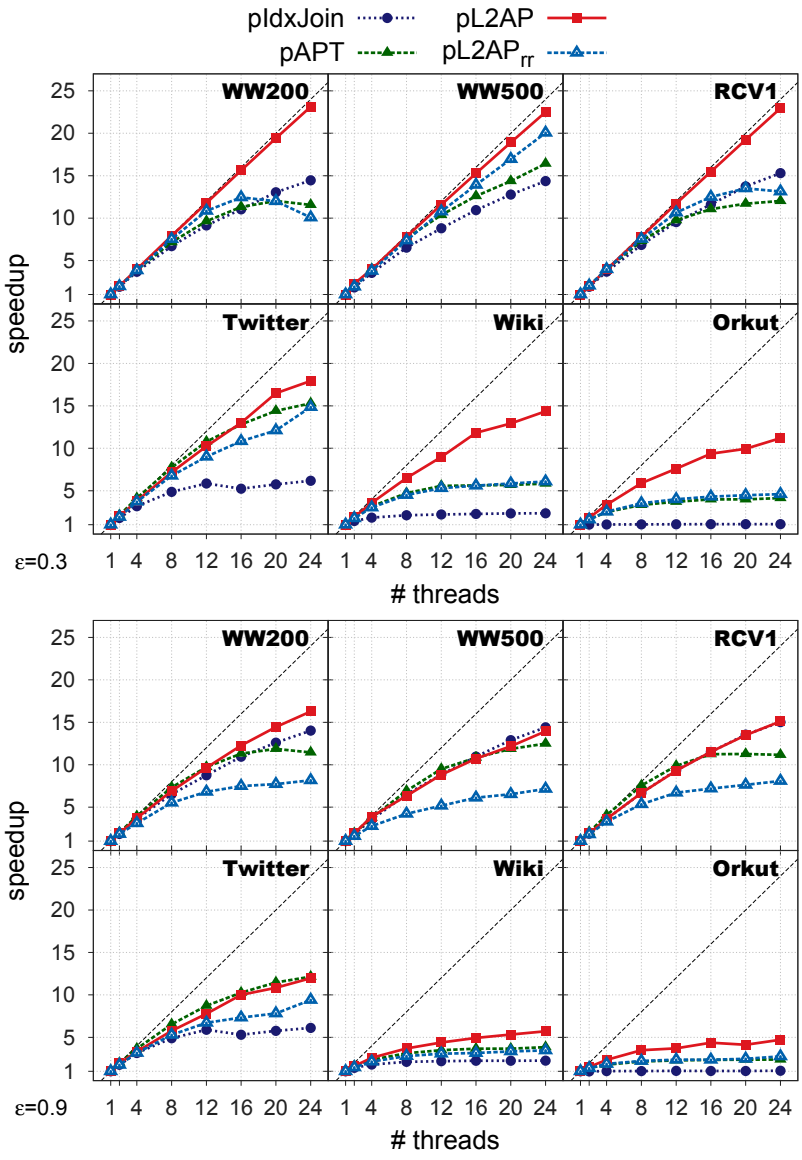

Figure 6.10: Strong scaling of parallel methods at $\epsilon = 0.3$ (top) and $\epsilon = 0.9$ (bottom).

Strong scaling

Figure 6.10 shows the strong scaling results from our experiments. The amount of work `pL2AP` does when processing each query increases as the threshold ϵ decreases. At high values of ϵ, many of the objects never become candidates for a query due to the idx and rs bounds in our method, and `pL2AP` is able to quickly dismiss candidates. For example,

the size of the candidate list when $\epsilon = 0.9$ is 0.5–4.0% of the candidate list size when $\epsilon = 0.3$ for text datasets. As a result, the cache locality improvements in `pL2AP` are not as beneficial, resulting in less pronounced scaling at $\epsilon = 0.9$. On the other hand, `pL2AP` shows linear scaling at $\epsilon = 0.3$ for text datasets. While its scaling is not as dramatic for network datasets, `pL2AP` still exhibits very strong scaling, in most cases better than the other baselines.

It is interesting to note that `pAPT` and $\texttt{pL2AP}_{\texttt{rr}}$ both scale poorly above twelve threads on text datasets. This may be an indication of thrashing, which is causing threads to waste time waiting for cache lines to be fetched from main memory.

Load balance

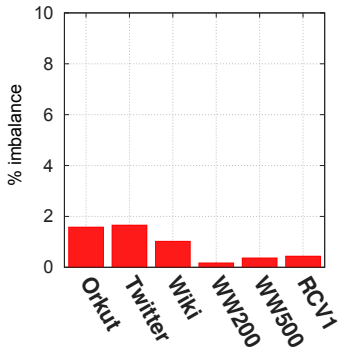

Figure 6.11: Load imbalance in `pL2AP`.

In order to test the effectiveness of the dynamic task partitioning approach in `pL2AP`, we measured the amount of time each thread spent searching for neighbors. Figure 6.11 shows the percent load imbalance averaged over all ϵ values, in experiments with consistent parameters ($nt = 24$, $h = 2^{13}$, $\eta = 25K$, and $\zeta = 1M$), for the six datasets. Load imbalance is computed as $100(t_{max}/t_{mean} - 1)$, where t_{max} and t_{mean} are the maximum and mean search times among the threads. Our method shows little imbalance between the threads, much less than 1% for text datasets and less than 2% for network datasets.

6.4 Parallel Cosine k-NNG Construction

We will start our discussion with an analysis of `L2Knng` and present some improvements to its serial execution, and then introduce `pL2Knng`, our parallel method for cosine k-NNG construction.

6.4.1 Serial Improvements in L2Knn

`L2Knng` execution consists of two phases. First, in the *approximate graph construction* phase, `L2Knng` finds an initial k neighbors for each of the objects in D by calling `L2Knng-a`. The minimum neighborhood similarities in each of the neighborhoods of the approximate graph are then used as pruning thresholds in the *filtering* phase, which outputs the *exact* nearest neighbor graph. `L2Knng-a` constructs the approximate graph in two steps. First, in the *initial graph construction* (IC) step, neighbors that are more likely to be in the exact k-NNG are chosen based on shared features with high weight. Then, a number of *graph enhancement* (GE) steps are executed which attempt to improve the quality of the neighborhoods by finding closer neighbors among the neighbors of the current neighbors. Algorithm 13 gives an overview of this process.

Algorithm 13 Computation flow in the L2Knng algorithm.

1: **function** L2KNNG($D,\ k,\ \gamma,\ \mu$)
▷ Begin `L2Knng-a`
2: $\hat{\mathcal{N}} \leftarrow IC(D,\ k,\ \mu)$
3: **for each** $i = 1, 2, \ldots, \gamma$ **do**
4: $\hat{\mathcal{N}} \leftarrow GE(D,\ k,\ \mu,\ \hat{\mathcal{N}})$
▷ End `L2Knng-a`
5: $\mathcal{N} \leftarrow Filter(D,\ k,\ \hat{\mathcal{N}})$
6: **return** $\mathcal{N}$

At a very high level, each of the steps in the `L2Knng-a` execution is composed of the following tasks, which are shown in Algorithms 14 and 15 and will be detailed later in the discussion. Input data or the current neighborhoods are sorted and indexed to facilitate the search for neighbors (*sort*). Then, for each query object, a candidate list of potential neighbors is selected (*sel*) that may improve the current neighborhood. Data associated with the query object is optionally entered into a data structure that can facilitate fast dot-product computations or pruning (*ins*). Then, dot-products are

computed between the query and each of the chosen candidates (*sim*), skipping some of the candidates whose similarity has already been previously computed. Finally, some of the neighborhoods are updated (*upd*) with computed similarities that improve them.

Algorithm 14 Initial graph construction in L2Knng-a.

```
1: function IC(D, k, μ)
2:    Create inverted index of D                                  ▷ srt
3:    Sort vectors in D and inverted index lists                  ▷ srt
4:    for each i = 1, 2, ..., |D| do
5:       Choose μ candidates for the ith object                   ▷ sel
6:       Hash the ith object                                      ▷ ins
7:       Compute similarities of d_i with all μ candidates        ▷ sim
8:       Update Γ_i and candidate neighborhoods                   ▷ upd
9:    N̂ = ⋃ Γ_i
10: return N̂
```

Algorithm 15 Graph enhancement in `L2Knng-a`.

```
1: function GE(D, k, μ)
2:    Create A, sparse matrix version of N̂                        ▷ srt
3:    Create inverted index of A                                  ▷ srt
4:    Sort vectors and inverted lists in A                        ▷ srt
5:    for each i = 1, 2, ..., |D| do
6:       Choose μ candidates for the ith object                   ▷ sel
7:       Hash the ith object                                      ▷ ins
8:       Compute similarities of d_i with all μ candidates        ▷ sim
9:       Update Γ_i and candidate neighborhoods                   ▷ upd
10:   N̂ = ⋃ Γ_i
11: return N̂
```

In an effort to gauge where the algorithm spends most of its time, we instrumented the `L2Knng` code with timers for each of the tasks. Table 6.4 shows the percent of the overall execution time in each phase taken by each of the tasks in the initial construction and graph enhancement phases, when searching for 10, 100, and 500 nearest neighbors in three datasets described in Section 2.3. In each of the experiments, we only executed one round of neighborhood enhancements ($\gamma = 1$) and chose candidate list sizes that would lead to average recall of at least 95%, i.e., `L2Knng-a` finds most of the nearest neighbors for each object. The last column in the table (*perc*) shows the percent of the overall `L2Knng-a` execution taken up by the current phase (IC or GE) of the algorithm. The results of this experiment show that `L2Knng-a` spends the majority of its execution

Table 6.4: Percent of the computation time for different sections of the approximate graph construction.

initial construction

dataset	**k**	*sort*	*sel*	*ins*	*sim*	*upd*	*perc*
RCV1	10	3.17	5.57	0.16	**88.04**	3.07	78
RCV1	100	4.44	5.70	0.26	**80.30**	9.30	39
RCV1	500	1.11	5.27	0.06	**83.48**	10.07	57
WW200	10	10.23	2.00	0.43	**86.45**	0.89	83
WW200	100	4.60	1.59	0.19	**92.02**	1.60	51
WW200	500	1.46	1.46	0.07	**94.82**	2.19	52
WW500	10	**24.07**	0.94	1.15	**73.06**	0.78	69
WW500	100	7.92	0.91	0.31	**89.57**	1.29	52
WW500	500	2.46	0.82	0.10	**94.77**	1.84	53

graph enhancement

dataset	**k**	*sort*	*sel*	*ins*	*sim*	*upd*	*perc*
RCV1	10	1.74	**20.59**	3.05	**69.54**	5.08	22
RCV1	100	2.65	**20.98**	0.26	**72.29**	3.82	61
RCV1	500	3.03	**26.84**	0.06	**66.64**	3.42	43
WW200	10	0.41	6.49	4.18	**87.19**	1.72	17
WW200	100	0.38	4.21	0.20	**94.35**	0.85	49
WW200	500	0.64	4.20	0.07	**94.15**	0.94	48
WW500	10	0.27	3.97	5.01	**89.52**	1.24	31
WW500	100	0.37	2.38	0.33	**96.25**	0.67	48
WW500	500	0.59	2.44	0.11	**96.03**	0.84	47

The table shows, for the initial graph construction and neighborhood enhancement phases of the `L2Knng-a` method, the percent of execution time of different tasks within each phase discussed in Section 6.4.1. The *perc* column shows the percent of the overall `L2Knng-a` execution taken up by the current phase of the algorithm. For each experiment, tasks taking up a significant portion of the execution time are highlighted in bold.

time computing similarities between query and candidate objects. Indexing and sorting when k is small and candidate selection can also account for a significant portion of the execution time. While graph enhancement takes up less time for small values of k, it accounts for almost half of the overall execution for larger k values.

Given these observations, we focused our efforts to improve `L2Knng-a` on the similarity computation, sorting, and candidate selection tasks. In the following sections we will detail each of the `L2Knng-a` tasks and our proposed improvements.

Index and sort

`L2Knng-a` chooses candidates in the IC phase by matching objects with common high weight features. To facilitate this search, it sorts the entries in each object vector and in each inverted index list in decreasing weight order. Then, it selects candidates for a query object by iterating through the inverted index lists associated with its highest weight features.

Since only μ candidates are selected for each query object, it is not necessary to fully sort all entries of the object vectors and inverted lists. With high probability, each inverted list will contain more than two entries (one entry will be associated with the query object). Thus, as an enhancement to `L2Knng-a` we propose sorting only the top-μ values in each vector and inverted list. For each vector and inverted list with lengths greater than μ, we first apply a select procedure [87], which partitions the list such that the leading μ values are greater or equal to the remaining values, and then sort only the leading μ values. This improvement reduces the complexity of sorting a list from $O(l \log l)$, where l is the size of the list, to $O(l + \mu \log \mu)$, and can be beneficial when μ is small or for datasets with very long vectors or inverted lists.

In each GE phase, `L2Knng-a` chooses candidates by matching neighbors and neighbors' neighbors with high similarity values. It first creates a sparse matrix version of the current approximate neighborhood graph, $\mathbf{A}$, such that the ith row of $\mathbf{A}$ corresponds to the k-neighborhood of the ith object. It then sorts the entries in each row in non-increasing value order and creates an inverted index for $\mathbf{A}$. `L2Knng-a` then selects candidates for a query object by iterating through rows in $\mathbf{A}$ associated with those objects that are the closest neighbors of the query, i.e., the column IDs of the leading entries in the sorted version of the row in $\mathbf{A}$ associated with the query.

The inverted index of $\mathbf{A}$ is used to identify objects whose similarity with the query has been previously computed. Additionally, for those query objects with less than μ candidates after the initial selection process, `L2Knng` further iterates through neighborhoods of objects that have the query object as their neighbor, in decreasing order of their similarity with the query. We call this process reverse candidate selection. The inverted list in $\mathbf{A}$ associated with the query contains the list of all objects that have the query in their neighborhood. `L2Knng-a` sorts the inverted lists in $\mathbf{A}$ in decreasing value order. In our experiments, we have found reverse candidate selection rarely improves

and can often degrade GE performance. Thus, in `pL2Knng`, we do not create an inverted index for **A** and only sort its row entries.

Candidate selection

In the IC phase, `L2Knng-a` selects candidates by iterating through two inverted lists at a time associated with the highest values in the query vector. Algorithm 16 describes this procedure. The function *nextList* provides the inverted list associated with the next smaller value in q. The function *nextCand* provides the next candidate in the chosen list, skipping the query object and any other objects that have already been selected. `L2Knng-a` uses an accumulation data structure to both track whether an object has already been selected as a candidate and to compute its partial dot-product with the query, which is denoted by $\langle \mathbf{q}, \mathbf{a}^{\leq} \rangle$ in Algorithm 16. Given two potential candidates c_a and c_b, `L2Knng-a` chooses c_a only if its partial dot-product with the query considering features already processed is greater than that of c_b.

Algorithm 16 Candidate selection in the IC phase of `L2Knng-a`.

```
1: function SelectCandidatesIC(D, q, μ)
2:   A ← nextList(q), B ← nextList(q), C = ∅
3:   while |C| < μ and A ≠ ∅ and B ≠ ∅ do
4:     if A = ∅ or B = ∅ then
5:       Choose candidates only from the remaining list
6:     a ← nextCand(A), b ← nextCand(B)
7:     if ⟨q, a^≤⟩ > ⟨q, b^≤⟩ then
8:       C ← C ∪ a
9:       A ← A \ a
10:      A ← nextList(q) if A = ∅
11:    else
12:      C ← C ∪ b
13:      B ← B \ b
14:      B ← nextList(q) if B = ∅
15:  end while
16: return C
```

We have improved candidate selection in the IC phase of `L2Knng-a` by simplifying the candidate choice condition (line 7 of Algorithm 16) to $d_{q,f(A)}d_{a,f(A)} < d_{q,f(B)}d_{b,f(B)}$, where $f(A)$ is the feature ID of list A, and $d_{i,j}$ is the value of the jth feature in the ith object. This simplification keeps the original intent in the selection and has not shown decreased quality performance in experiments. Instead, the efficiency of this step

is increased by removing the need to compute partial dot-products. We use a bitvector data structure to track candidates that have already been selected, which uses less cache memory and may also help increase performance.

Algorithm 17 Candidate selection in the GE phase of `L2Knng-a`.

```
1: function SelectCandidatesGE(D, q, μ)
2:   a ← nextNeighbor(q), A = neighborhood(a), C = ∅
3:   while |C| < μ and A ≠ ∅ do
4:     c ← nextCand(A)
5:     if sim(a, c) ≥ sim(q, a) then
6:       C ← C ∪ c
7:     A ← A \ c
8:     if A = ∅ then
9:       a ← nextNeighbor(q)
10:      A ← neighborhood(a)
11:  end while
12: return C
```

In the GE phase of our method, candidates are selected by iterating through neighbors' neighborhoods, one at a time, as shown in Algorithm 17. The *nextNeighbor* function selects the neighbor a with the next smaller similarity value in the query's neighborhood. The list returned by the *neighborhood* function is the row in $\mathbf{A}$ associated with the selected neighbor, which contains its neighbors, sorted in decreasing similarity value order. While iterating through these neighbors, candidates are only accepted if their similarity value is greater than the similarity between a and the query. We have not made changes to the selection process in this phase of `L2Knng-a`.

Query insertion and similarity computation

Since `L2Knng-a` computes the similarity of a query vector with many different candidate vectors, it creates a dense version of the query vector, stored in memory in an array, which can be reused to compute μ dot-products. Each dot-product is computed as a sparse-dense vector dot-product, by iterating through the non-zero values of the candidate vector and looking up values of the query vector in the array. Given an array Q representing the dense version of $\mathbf{d}_q$, the dot-product $\langle \mathbf{d}_q, \mathbf{d}_c \rangle$ can be computed as,

for each $j = 1, \ldots m$ **s.t.** $d_{c,j} > 0$ **do**

$$s \leftarrow s + d_{c,j} Q[j]$$

We call this dot-product computation strategy, along with all the improvements discussed thus far, the *base* `L2Knng-a` strategy.

As computing dot-products takes up the most time in the `L2Knng-a` execution, we tried several other strategies for executing this operation. Sparse vectors are represented in `L2Knng-a` as two arrays, one containing feature IDs of non-zero features in the vector, and the other containing the values for the corresponding features. In one strategy, trying to take advantage of vectorization capabilities of modern hardware, we first packed the longer of the two vectors into a temporary array corresponding to the non-zero features in the shorter vector, allowing the dot-product to be executed as a dense vector dot-product between the temporary array and the values array of the shorter vector. Dot-products can also be computed in a sparse-sparse fashion, by traversing the feature ID arrays of both sparse vectors simultaneously and executing a multiply-add operation when encountering matching feature IDs. Finally, we tried using the query vector mask-hashing technique described in Section 6.2.2 to speed up dot-products in `L2Knng-a`. Mask-hashing uses the same sparse-dense computation strategy as `L2Knng-a` but replaces the vector with a hash table designed for fast in-order look-up of features that may exist in the query vector while reducing the amount of cache memory necessary to store the query vector values. In our experiments, none on of the new dot-product computation strategies improved the performance under a wide range of execution parameters. One disadvantage of these strategies is that they require maintaining a version of the sparse vectors with entries sorted in feature ID order, which is not necessary in the base strategy.

As another strategy to improve similarity computation efficiency, we added prefix ℓ^2-norm based pruning to the dot-product computation step, which we have found to be a very effective tool for eliminating false positive candidates in the `L2Knng` filtering framework. Algorithm 18 describes this strategy, which we call *prune*. The symbol R represents another dense vector that contains query prefix ℓ^2-norms at the associated non-zero query features, $R[j] = \|\mathbf{d}_q^{<j}\|$ iff $d_{q,j} > 0$. In addition to the candidate vectors value array, we also keep an array with prefix norms associated with those features,

which can be used each time the candidate is involved in a similarity computation. The symbol σ represents the minimum similarity of either the query of candidate neighborhoods, $\sigma = \min(\sigma_{d_q}, \sigma_{d_c})$. The pruning step can lead to early termination of dot-product computations, and can also reduce the number of neighborhood update attempts.

Algorithm 18 Similarity computation with pruning in `L2Knng-a`.

```
1: function SIM(d_c, Q, R, σ)
2:     s ← 0
3:     for j = 1, ... m s.t. d_{c,j} > 0 do
4:         if d_{c,j} > 0 then
5:             s ← s + d_{c,j} Q[j]
6:             if s + ||d_c^{<j}|| R[j] < σ then
7:                 return null
8: return s
```

Neighborhood updates

After computing each similarity between a query and a candidate vector, `L2Knng-a` updates the query and candidate neighborhoods if the similarity value can improve those neighborhoods. We improved this step in several ways. First, we separated the similarity computation from the update steps in the algorithm, allowing `L2Knng-a` to compute all μ similarities before inserting any update. This improves cache locality during similarity computation. Second, we update neighborhoods in two stages, inserting all updates into the query neighborhood before updating other neighborhoods, which further improves cache locality. Third, since at most the top-k of the μ computed similarities have any potential of improving the query neighborhoods, we fist apply a k-select procedure on the candidate list, and then attempt to update the neighborhood with only the first k items in the list. We call this strategy *select*. It has the potential to further improve cache locality of the query neighborhood update stage, especially for large μ and/or k.

6.4.2 pL2Knn

Algorithm 19 described our parallel k-NNG construction method, `pL2Knng`. Our method follows the same computation strategy as `L2Knng`, incorporating the improvements described in Section 6.4.1. In addition, `pL2Knng` uses a cache-tiling strategy similar to the

one in `pL2AP` (see Section 6.2.2). The method splits the inverted index into several consecutive sections, called *tiles*, and each index is used in turn to find neighbors. The size of each tile is dynamically chosen based on a maximum number of objects parameter ν and a maximum number of non-zeros ζ. After processing each index tile, `pL2Knng` uses the block completion strategy in `L2Knng` to complete the search for all the objects in the tile, and then discards the tile. This step improves the minimum neighborhood similarities of un-processed objects in the neighborhood graph represented by $\hat{\mathcal{N}}$. As a results, `pL2Knng` then updates the object processing order of un-processed objects, improving the index reduction and pruning potential in filtering the following tile.

Algorithm 19 The `pL2Knng` algorithm.

function pL2KNN(D, k, ζ, ν, η)
  Set processing order for features
  $\hat{\mathcal{N}} \leftarrow pL2KNN\text{-}a(D,\ k)$
  Set object processing order given $\hat{\mathcal{N}}$
  $z \leftarrow 0,\ r \leftarrow 0,\ i \leftarrow 1,\ \mathcal{I} \leftarrow \varnothing$
  while $i \leq n$ **do**
    $k \leftarrow i$
    for each $i = k, \ldots, n$ **do** ▷ Identify next tile
      $S \leftarrow \text{FindIndexSplit}(\mathbf{d}_i, \sigma_{d_i})$
      $z \leftarrow z + nnz(\mathbf{d}_i^{>})$
      $r \leftarrow r + 1$
      if $z \geq \zeta$ **or** $r = \nu$ **then**
        $i \leftarrow i + 1$
        break
    for each $q = k, \ldots, i$ **in parallel do** ▷ Create tile index $\mathcal{I}$
      $\text{Index}(\mathbf{d}_q, \mathcal{I}, S, \sigma_{d_q})$
    for each $l = k, \ldots, n$, **in increments of** η **do** ▷ Filter
      for each $q = l, \ldots, \min(l+\eta\ -1,\ \text{n})$, **in parallel do**
        $\mathbf{c}_q \leftarrow \text{GenerateCandidates}(\mathbf{d}_q, \mathcal{I}, k)$
        $\text{VerifyCandidates}(\mathbf{d}_q, \mathbf{c}_q, \mathcal{I}, \hat{\mathcal{N}}, k)$
    $\mathcal{I} \leftarrow \varnothing$
    Update un-processed object processing order given $\hat{\mathcal{N}}$
  end while
return $\hat{\mathcal{N}}$

Unlike `pL2AP`, where individual threads can collect output data for objects they process, which can be easily merged at the end of the execution, similarities in `pL2Knng` are used to update the query and candidate neighborhoods. This poses the risk of resource contention, as multiple threads may try to update the same neighborhood at

the same time. One solution could be for each thread to lock a *mutex* associated with the neighborhood before updating it. However, with the number of neighborhoods in the $n = 10^5$–10^7 range, and number of potential neighborhood updates being as high as $0.5n(n-1)$, this strategy would be very slow. Additionally, threads would have to wait while a neighborhood they needed to update was being updated by another thread, which would cause further slowdowns. Instead, we have devised a lock-free strategy for updating query and candidate neighborhoods in parallel, which we use in both the `pL2Knng-a` and `pL2Knng` algorithms.

Our methods process queries in tiles up to η objects at a time (line 18), which we call query tiles. We allocate memory for up to η accumulators, which are used by threads to store their search results for each object they process in the query tile. Within the tile, each thread is dynamically assigned a few objects at a time to process. The neighborhood of each processed object can be safely updated with the results. Contention occurs only when updating candidate neighborhoods. We assign each thread a sequential block of n/nt candidate objects whose neighborhoods they are responsible to update. After processing each object and updating its neighborhood, threads rearrange the accumulator array such that it stores candidate updates sequentially. They also mark the start and end offsets in the accumulator for each thread's set of assigned candidates. Then, after a query tile has been processed, threads iterate through their assigned sections of all the accumulators in the tile, updating neighborhoods for candidates they are responsible for.

6.5 Experimental Evaluation for Parallel Cosine k-NNG Construction

Our experiment results are organized along two directions. First, we present results from evaluating our parallel approximate method, `pL2Knng-a`. We compare our method's accuracy and efficiency against approximate baselines, and then study the strong scaling characteristics of the approximate methods under comparison. Second, we present results from evaluating our exact method, `pL2Knng`. We measure serial efficiency improvements compared to the original `L2Knng` algorithm, study our method's sensitivity to parameter choices, compare the efficiency and strong scaling characteristics of `pL2Knng`

with parallel and approximate baselines, and study load imbalance in our method. For all experiments, we used three real-world text datasets for evaluation, RCV1, WW200, and WW500, which are described in Section 2.3.

6.5.1 Baseline Approaches

We compare our methods against the following baselines.

- `pKIdxJoin` is a straight-forward baseline similar to *IDX* in [37] and `kIdxJoin`, described in Section 5.2. The method uses similar cache-tiling as `pL2Knng`, but does not use any pruning when computing similarities. For each block of queries, `pKIdxJoin` sequentially retrieves a block of objects to search against and indexes all their values. Threads then share the index to compute similarities, via accumulation, of each assigned object in a query tile against all indexed objects, retaining the top-k matches for each object.
- *Greedy Filtering* is an approximate k-NNG construction method proposed by Park et al. [37], which we described in Section 5.2. We have created a shared memory parallel version of *Greedy Filtering*, which we call *pGF*, using the same thread cooperation strategy as in `pL2Knng-a`. Threads first work together to index enough high-weight features for each object to ensure μ candidate neighbors have at least one feature in common with each input object. Then, they dynamically split the work of computing similarities of each object in an inverted list against all other objects in the list. We adopt the same neighborhood update strategy as in `pL2Knng`. Each thread updates the neighborhood of an assigned query object as soon as it has finished computing the similarity with a candidate object. Threads synchronize at the end of each inverted index list, reading computed similarities by all threads in order to update neighborhoods for an assigned block of objects.
- *NN-Descent* is a shared memory parallel approximate k-NNG construction method designed by Dong et al. [36], which we described in Section 5.2.

Locality sensitive hashing (LSH) has been a popular method for top-k search, but we have found that it does not in general perform well in the k-NNG construction setting when one requires high average recall. Both *Greedy Filtering* and *NN-Descent* have been shown to outperform LSH in this setting, for k typically ≥ 10. Moreover, `pL2Knng`

Table 6.5: Approximate graph construction similarity computation and pruning strategies.

k	nt	base	select	prune	ps	psi1	psi2
				RCV1			
10	1	**238.22**	239.45	282.56	292.95	280.89	314.82
100	1	**425.62**	435.53	503.57	532.01	502.48	519.58
500	1	**1738.29**	1750.71	2017.61	2001.87	2216.25	2201.37
10	16	25.78	**25.77**	30.05	30.37	30.52	31.50
100	16	95.31	**47.58**	57.99	58.93	54.66	55.40
500	16	663.22	247.25	278.98	283.25	224.44	**202.06**
				WW200			
10	1	**520.49**	545.50	575.09	577.69	545.71	556.06
100	1	**2184.82**	2589.67	2377.12	2420.05	2411.12	2443.35
500	1	**8183.49**	8947.30	8607.76	10542.93	9563.67	9283.00
10	16	70.78	**61.08**	73.57	62.50	71.92	61.09
100	16	296.13	**253.60**	314.95	315.60	266.58	262.15
500	16	1187.05	**934.32**	1244.82	1094.21	1030.59	1034.11
				WW500			
10	1	**98.78**	99.04	102.44	100.29	100.81	103.70
100	1	536.53	528.34	544.50	569.75	**493.04**	529.31
500	1	2126.66	2107.47	2232.83	2244.38	**2073.01**	2192.98
10	16	**13.25**	13.28	13.72	13.87	13.68	13.40
100	16	73.56	73.51	77.02	77.10	**70.35**	70.57
500	16	308.14	309.56	319.98	321.38	**252.87**	258.61

The table shows execution times for our `pL2Knng-a` approximate graph construction algorithm, in seconds, under the different similarity computation and pruning strategies described in Section 6.4.1, for neighborhood sizes $k \in \{10, 100, 500\}$ and number of threads $nt \in \{1, 16\}$, for three different datasets. For each configuration, the best execution time is highlighted in bold.

significantly outperforms *Greedy Filtering* and *NN-Descent* in both serial and parallel execution environments. As a result, we have chosen not to compare against LSH in this work.

6.5.2 Evaluation of Approximate Methods

Strategy comparison

In Section 6.4.1, we have presented several strategies for improving our approximate k-NNG construction method. We tested each strategy on three test datasets, using both our serial (`L2Knng`, $nt = 1$) and parallel algorithms (`pL2Knng`, $nt = 16$), and present the results in Table 6.5, for $k \in \{10, 100, 500\}$. In addition to the *base*, *select*, and

prune strategies described in Section 6.4.1, we tested three additional scenarios. The *ps* strategy combines the prune and select strategies. Checking pruning bounds can be an expensive operation, which is only beneficial if it results in pruning. One option would be to only check the bounds some of the time, e.g., for the first few traversed non-zeros in the candidate vector. We tested two such scenarios, where *psi1* and *psi2* use the same strategy as *ps*, but check the pruning bounds only during the first 20 and 250 accumulation operations, respectively.

The results show that the *select* strategy was beneficial in the parallel setting, incurring up to 2.68x less execution time than the *base* strategy, but did not improve serial execution. On the other hand, the *pruning* strategy, which checked the bound after each accumulation operation, was in general slower than all the other strategies. Partial pruning strategies *psi1* and *psi2* were the most effective for large k values. Based on these results, we chose *psi1* as the strategy under which we executed all experiments presented in the remainder of this section.

Serial improvement comparison

Table 6.6: Efficiency improvement in `L2Knng`.

method	versus	k=10	25	50	75	100
		WW200				
`L2Knng`	`L2Knng`*	**1.63**	**1.68**	**1.71**	**1.70**	**1.70**
`L2Knng-a`	`L2Knng-a`*	1.10	1.26	1.18	1.21	1.15
`pL2Knng-a`	`pL2Knng-a`*	**2.15**	**1.95**	1.45	1.48	1.43
		WW500				
`L2Knng`	`L2Knng`*	1.49	**1.60**	**1.62**	**1.73**	**1.69**
`L2Knng-a`	`L2Knng-a`*	1.31	1.27	1.35	1.26	1.31
`pL2Knng-a`	`pL2Knng-a`*	**2.74**	**2.42**	**1.96**	**1.66**	1.48
		RCV1				
`L2Knng`	`L2Knng`*	1.46	1.50	1.49	**1.54**	1.44
`L2Knng-a`	`L2Knng-a`*	1.09	1.15	1.18	1.23	1.39
`pL2Knng-a`	`pL2Knng-a`*	1.47	**1.54**	**1.63**	**1.58**	**1.61**

The table shows speedup values for each of our new `L2Knng`, `L2Knng-a`, and `pL2Knng-a` methods, which implement the improvements described in Section 6.4.1, versus the previous version of the same algorithm, noted with a star. Significant improvements are marked in bold.

We compared the search execution times of our new `L2Knng`, `L2Knng-a`, and `pL2Knng-a`

methods, which implement the improvements described in Section 6.4.1, versus the previous version of the same algorithm. We executed experiments for $k \in \{10, 25, 50, 75, 100\}$ on our three datasets, and we used $nt = 16$ threads when executing parallel methods. Table 6.6 shows the result for this set of experiments. Significant improvements (1.5x or above) of our enhanced methods versus their previous versions are marked in bold. The results show that the improvements are beneficial, especially for serial execution in `L2Knng` and parallel execution in `pL2Knng-a`, leading to speedups of up to 2.74x. In the remainder of this section, we will use the new versions of `L2Knng` and `L2Knng-a`, which include the improvements described in Section 6.4.1, in all experiments comparing execution of our parallel methods against a serial baseline.

Effectiveness comparison

The efficiency of all the approximate methods under consideration are dependent on the number of candidates they are allowed to consider for each object, μ. The larger the candidate pool is, the more likely the true neighborhood is found among the objects in the pool. We compared the recall and execution time of `pL2Knng-a` with other approximate baselines, given the same candidate list and neighborhood size parameters, μ and k. We tested each method, without changing any other parameters, given $\mu = k, 2k, \ldots, 10k$, on the RCV1 and WW500 datasets. We tested `pL2Knng-a` with $\gamma = 0$ (`pL2Knng-a`$_0$), which does not execute any iterative neighborhood updates, and with $\gamma = 3$ (`pL2Knng-a`$_3$). All methods were executed with $nt = 16$ threads.

Figure 6.12 plots recall versus execution time for our experiment results. For all methods, results for $\mu = k$ are marked with a "-" label, and those for $\mu = 10k$ with a "+" label. The best results are those points in the lower-right corner of each quadrant in the figure, achieving high recall in a short amount of time. We display results for $k \in \{50, 100\}$. Results for other k values showed similar trends.

The results are consistent with the same experiment we executed on the serial version of these methods, detailed in Section 5.2.2. Methods generally exhibit higher recall and higher execution time for larger μ values. *NN-Descent* took considerably more time than most other methods to complete the graph construction. `pL2Knng-a`$_0$ takes much less time to execute than *pGF* and, given large enough μ, can achieve similar or higher recall. Both `pL2Knng-a` and *pGF* require larger μ values than *NN-Descent* to achieve

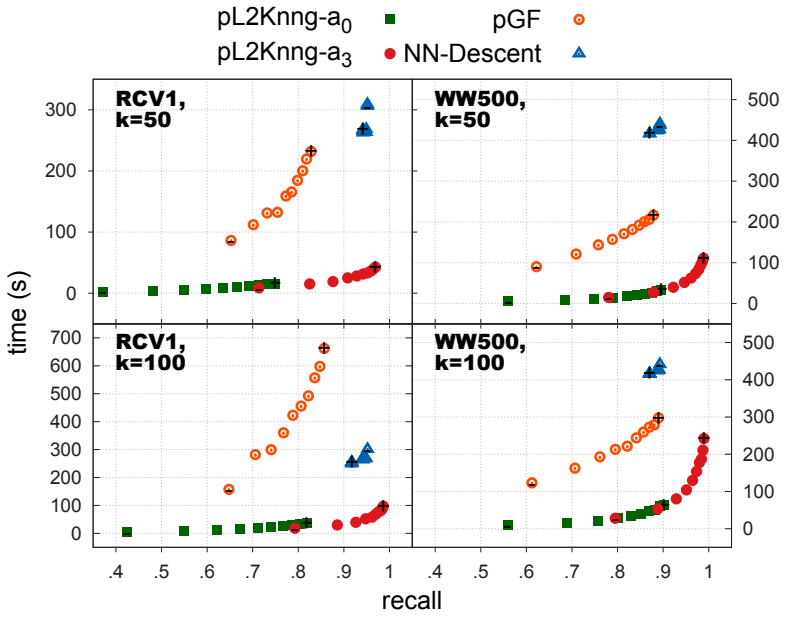

Figure 6.12: k-NNG construction effectiveness comparison.

high recall. Yet, *NN-Descent* does not improve much as μ increases. `pL2Knng-a`$_3$ is able to outperform both competitors, with regards to both time and recall, for large enough μ.

Efficiency comparison

Figure 6.13 displays the results of our parallel approximate k-NNG construction method efficiency comparison experiments. Execution times are displayed on the left, and speedup over the best serial approximate method (`L2Knng-a`) is displayed on the right. We executed each approximate method under a wide range of parameters and report the smallest time for which a minimum recall value of 0.95 was achieved. We were not able to achieve high enough recall for *NN-Descent* for the WW200 dataset for $k \in \{10, 25\}$. Therefore, the graph contains no bars for *NN-Descent* for those results.

Our method, `pL2Knng-a`, was more efficient than all baselines in all experiments and achieved 8–13x speedup using 16 cores over the serial version, showing that our lock-less neighborhood update strategy is effective. We used the same neighborhood update strategy in *pGF*, yet its efficiency degrades quickly with increasing k, likely due to its candidate selection strategy, which can lead to many similarities being computed

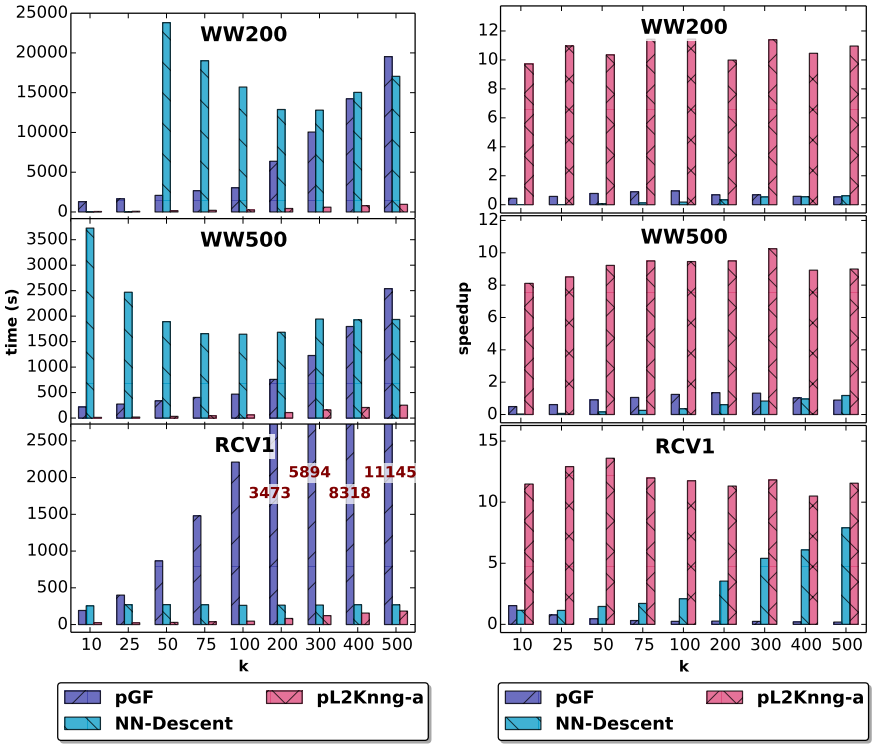

Figure 6.13: Approximate k-NNG construction efficiency comparison.

multiple times during the search. *NN-Descent* performed poorly on the WW datasets, especially for low values of k, which required setting the candidate list size μ very high to achieve the minimum required recall. Surprisingly, it performed much better on the RCV1 dataset, coming within 1.5x of the performance of `pL2Knng` for $k = 500$. *NN-Descent* computes similarities one at a time, as a sparse-sparse vector dot-products. Object vectors in RCV1 are fairly short in comparison with those in the WW datasets, which may explain the difference in performance.

Strong scaling

Figure 6.14 shows the strong scaling results from our approximate methods, for $k = 10$ (left) and $k = 100$ (right). Parameters were chosen for each method to achieve a minimum recall of 0.95. We used query tile size $\eta = 25$k for `pL2Knng-a`. Our method displays a consistent scaling pattern, achieving slightly more than 8x speedup using 16 threads. In general, *pGF* scaled worse than `pL2Knng-a`. While *NN-Descent* showed

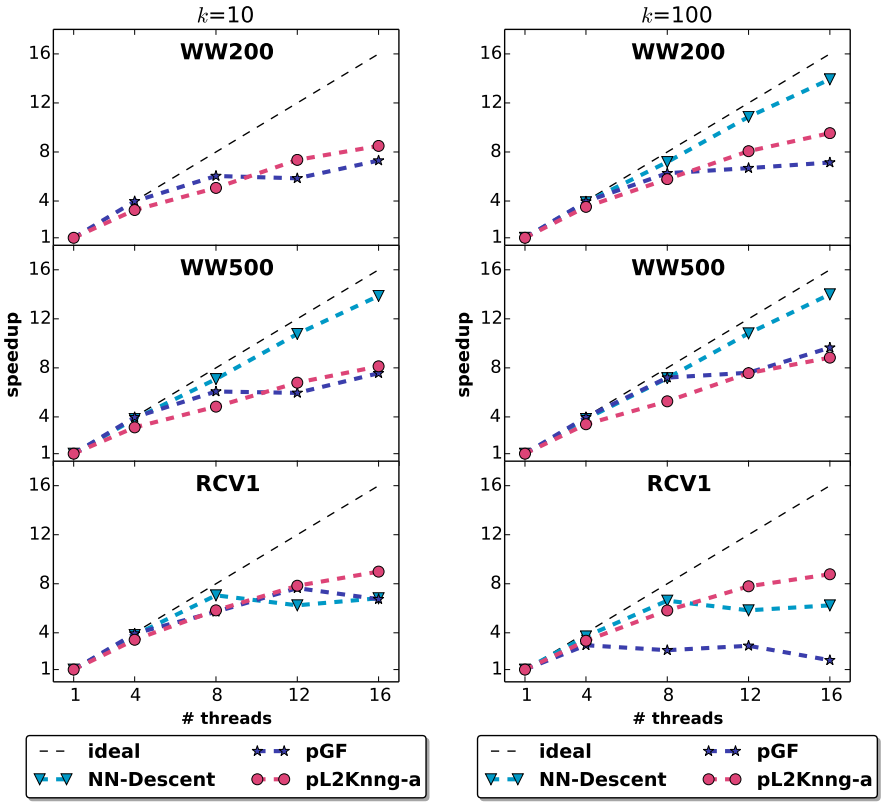

Figure 6.14: Strong scaling of approximate k-NNG construction methods.

better scaling characteristics for the WW datasets up to 16 threads, its execution time is orders of magnitude slower. For the RCV1 dataset, *NN-Descent* did not scale better than 8x.

6.5.3 Evaluation of Exact Methods

Parameter sensitivity

The tiling parameters, η, ν, and ζ, affect the size of the data structures used when searching for neighbors, which should fit in the processor cache to achieve good performance. As a way to test the sensitivity of our exact k-NNG construction method to these input parameters, we executed experiments with all combinations of $\eta \in \{10\text{k}, 25\text{k}\}$ and $\zeta \in \{0.5\text{M}, 1\text{M}, 5\text{M}, 10\text{M}\}$. For all experiments, we set $\nu = \infty$, allowing the ζ parameter

Table 6.7: Parameter sensitivity analysis in pL2Knng.

k=10			k=100			k=500		
η	ζ	cmp	η	ζ	cmp	η	ζ	cmp
10k	0.5M	0.98	10k	0.5M	0.99	10k	0.5M	1.15
10k	1M	1.03	10k	1M	1.02	10k	1M	1.18
10k	5M	1.60	10k	5M	1.43	10k	5M	1.42
10k	10M	1.80	10k	10M	1.54	10k	10M	1.49
25k	**0.5M**	**0.95**	**25k**	**0.5M**	**0.98**	25k	0.5M	1.14
25k	1M	1.00	25k	1M	1.00	**25k**	**1M**	**1.00**
25k	5M	1.57	25k	5M	1.41	25k	5M	1.41
25k	10M	1.77	25k	10M	1.51	25k	10M	1.49

to solely decide the number of objects in each inverted index. Table 6.7 presents the results of these experiments, for $k \in \{10, 100, 500\}$, as speedup relative to the experiment result for $\eta = 25$k and $\zeta = 1$M.

Our method displays the best performance for small enough ζ values, when the inverted indexes can fit in the processor cache. Setting ζ very high can result in more than 1.5x slowdown as compared to an optimal setting. The choice of query tile size η does not seem to affect performance as much. For experiments detailed in the remainder of the section, we set parameters $\eta = 25$k and $\zeta = 1$M for pL2Knng.

Load balance

Table 6.8: Load imbalance in pL2Knng.

	time (s)				*imbalance*			
k	*IG*	*GE*	*CG*	*CV*	*IG*	*GE*	*CG*	*CV*
RCV1								
10	33.11	1.01	99.91	32.98	1.83	1.33	0.19	0.78
100	35.98	20.51	217.67	66.11	11.28	2.23	0.07	0.34
500	175.35	84.85	359.22	98.96	12.47	5.42	0.16	0.52
WW200								
10	74.60	6.02	1176.66	125.56	0.73	0.30	0.12	0.60
100	158.57	144.79	1955.26	165.52	4.15	0.30	0.11	1.59
500	667.56	536.71	2711.16	194.48	12.71	0.98	0.14	1.67
WW500								
10	11.96	2.15	175.49	10.46	0.21	0.09	0.14	1.06
100	39.87	35.81	301.42	12.91	2.71	0.11	0.22	1.70
500	171.55	142.41	422.11	18.82	9.41	0.49	0.15	1.57

In order to test the effectiveness of the dynamic task partitioning approach in `pL2Knng`, we measured the amount of time each thread spent searching for neighbors, in each of the parallel search regions of our method, initial graph construction (CG), graph enhancement (GE), filtering candidate generation (CG), and filtering candidate verification (CV). We executed the experiment for $k \in \{10, 100, 500\}$ on three datasets and present the result in Table 6.8. The left side of the table shows execution time for each of the parallel regions, while the right side shows the amount of load imbalance in the execution. Following DeRose et al. [88], we measure load imbalance percentage as,

$$\% \text{ load imbalance} = \frac{n}{n-1}\left(\frac{t_{max}}{t_{mean}} - 1\right),$$

where t_{max} and t_{mean} are the maximum and mean search times among the threads. The measure corresponds to the percentage of time that other threads, excluding the slowest one, are not engaged in useful work during the given parallel block. All experiments were executed with $nt = 16$ threads.

The results of our experiment show that our method exhibits good load balance in general, especially in the filtering stages. The IC stage, which takes a relatively small part of the overall execution (14% on average across our experiments), exhibits the most load imbalance, which is generally less than 13%. While the imbalance seems to increase with larger values of k for the IG and GE sections, it does not seem to be affected in the same way in the filtering stages, CG and CV.

Efficiency comparison & strong scaling analysis

Figure 6.15 displays the results of our parallel exact k-NNG construction method efficiency comparison experiments. Execution times are displayed on the left, and speedup over the best serial exact method (`L2Knng`) is displayed on the right. As `pKIdxJoin` does not use any pruning, its execution times are similar for all k values, affected in general only by sorting longer lists during the nearest neighbor identification for each object. However, even with 16 threads, it only achieves 1.5–4.5x speedup over our serial baseline. In contrast, `pL2Knng` significantly outperforms `pKIdxJoin` in every experiment and achieves 12.5–15.5x speedup over the serial baseline. This highlights both the effectiveness of the pruning in `pL2Knng`, and the ability of threads to cooperatively solve

the problem without contention or much additional overhead.

Figure 6.16 shows the results of our strong scaling analysis for $k \in \{10, 100\}$ using all three of our test datasets. In each experiment, we report the scaling of each method at $nt \in \{1, 4, 8, 12, 16\}$ over its own single threaded execution. The dashed line shows ideal scaling. Results show that `pL2Knng` has very good strong scaling characteristics, achieving up to 14x speedup, while `pKIdxJoin` is unable to scale better than 9x.

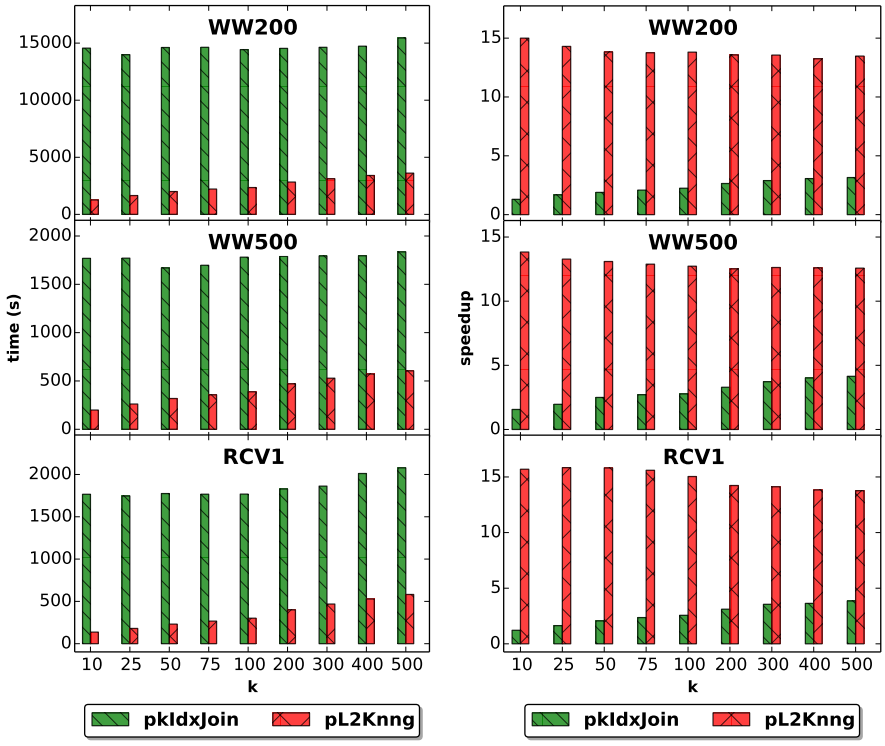

Figure 6.15: Exact k-NNG construction efficiency comparison.

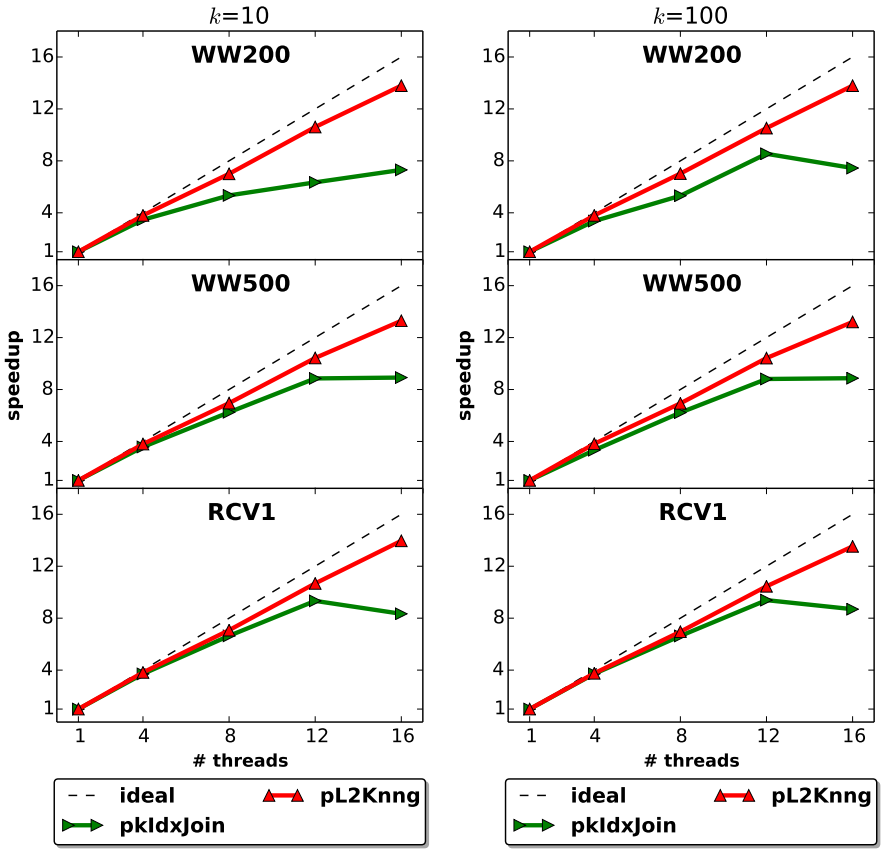

Figure 6.16: Strong scaling of exact k-NNG construction methods.

Chapter 7

Conclusion

Constructing nearest neighbor graphs for large sets of objects is a difficult problem, which, due to its high complexity, is often solved approximately, returning only some of the nearest neighbors for each object. In this thesis, we presented novel solutions for constructing exact nearest neighbor graphs, which contain all of the nearest neighbors, both on serial and shared memory parallel architectures.

In Chapter 4, we addressed the ϵ-NNG construction problem, which constructs a graph by finding, for each object in the set, all other objects with a similarity of at least some threshold ϵ. First, leveraging the Cauchy-Schwarz inequality in partial vector dot-products, we proposed new theoretic upper bounds on the cosine similarity of two vectors and proved their superiority to other bounds previously proposed in the literature. We then described how these bounds can be efficiently tested to allow pruning much of the search space when constructing the graph, and analyzed the performance of individual bounds. The newly introduced prefix ℓ^2-norm based bounds showed the most pronounced improvement on pruning performance and execution efficiency, allowing our method, `L2AP`, to construct the graph 2–13x faster than the best alternative, depending on the input threshold ϵ, on sets of 100k–3M objects representing documents or social network profiles. In particular, our method was able to find all pairs of pages with nearly identical links among 1.8M English Wikipedia pages in 10 seconds, using a single CPU core. `L2AP` was up to 1600x faster than a linear search for neighbors, which does not prune the search space. While baseline algorithms do not scale well as the similarity threshold decreases, `L2AP` was effective at both high and low similarity thresholds. In

many of the experiments, our exact graph construction method was able to outperform even approximate methods required to obtain at least 95% of the correct result.

Second, we showed that the filtering techniques we designed for cosine similarity can also be used to solve the ϵ-NNG construction problem for some length-variant similarity functions, such as the Tanimoto similarity. Our solution combined some existing filtering techniques, the bounds we designed for the cosine function, and a new class of length-based bounds into a cohesive method for constructing the neighborhood graph using the Tanimoto similarity function, named `TAPNN`. We proved the effectiveness of our new filtering bounds and showed that `TAPNN` significantly outperforms state-of-the-art baselines, for a range of ϵ values. Our method was up to 12.5x more efficient than the most efficient baseline and up to two orders of magnitude faster than a linear search. In particular, it was able to find all near-duplicate pairs among 5M chemical compounds in minutes, using a single CPU core.

In Chapter 5, we addressed the k-NNG construction problem, which constructs a graph by finding the k closest other objects for each object in the input set. We introduced a novel method for constructing the cosine k-NNG, which combined a fast algorithm for obtaining an initial approximate solution to the problem with a modified filtering framework. In this framework, we introduced several new pruning bounds specific to the k-NNG construction problem and data structures for efficiently constructing the graph. We performed an extensive evaluation of our algorithm, comparing against both exact and approximate baselines, across a variety of real-world datasets and neighborhood sizes. Our experiments revealed that our approximate k-NNG construction method, `L2Knng-a`, achieves high recall in less time than competing approximate methods, and is an order of magnitude more efficient than approximate baselines when building a graph that is at least 95% correct. Furthermore, our exact method, `L2Knng`, was 2.2–28.2x faster than exact baselines in our experiments.

In Chapter 6, we described shared memory parallel methods for both the ϵ-NNG [22] and the k-NNG [23] construction problems. We introduced a number of cache-tiling optimizations, which, combined with fine-grained dynamically balanced parallel tasks, allowed our methods to solve the problem up to two orders of magnitude faster than existing parallel baselines, on datasets with hundreds of millions of non-zeros. In particular, our parallel ϵ-NNG method, `pL2AP`, was able to construct the exact graph, using 24

cores, 1.5–232x faster than the best alternative. We tested our parallel k-NNG method, `pL2Knng`, using 16 cores, and found that it outperformed baselines by 3.0–12.9x. Both `pL2AP` and `pL2Knng` displayed good scaling characteristics, superior to those of their respective baselines. Furthermore, our parallel approximate k-NNG construction method, `pL2Knng-a`, was able to construct a graph containing at least 95% of the correct result 1.5–21.7x faster that previous methods.

The methods presented in this thesis can be extended in a number of potential future directions. First, to allow solving problems with hundreds of million of objects or more, efficient methods should be designed for distributed computing platforms and for cloud computing. A key issue in these environments is minimizing the communication overhead needed to solve the problem. Some of the filtering strategies we applied in the serial and shared memory parallel environments could be applied to intelligently partition the input data and work assignments among nodes in a way that can minimize communication, while simultaneously prune some of the search space.

The methods I presented in this thesis make the assumption that all input objects are present at the onset of execution and that their vector representations fit in memory. Moreover, data are assumed not to change. An interesting future direction would be to design persistent data structures that can enable both fast nearest neighbor graph construction and record updates. It would be interesting to investigate filtering techniques that can be applied to make neighborhood and inverted index data structure updates efficient.

Finally, another interesting direction would be to design graph construction methods that take advantage of hardware accelerators, such as Intel Many Integrated Core (MIC) coprocessors or graphical processing units (GPUs). MIC and GPU accelerators present different challenges for designing scalable parallel methods, but can make it feasible to solve very large problems, which is very desirable given the large volume of Big Data today.

References

[1] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. In *Selected papers from the sixth international conference on World Wide Web*, pages 1157–1166, Essex, UK, 1997. Elsevier Science Publishers Ltd.

[2] Taher H. Haveliwala, Aristides Gionis, and Piotr Indyk. Scalable techniques for clustering the web. In *Proc. of the WebDB Workshop*, pages 129–134, 2000.

[3] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Detectives: Detecting coalition hit inflation attacks in advertising networks streams. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 241–250, New York, NY, USA, 2007. ACM.

[4] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 271–280, New York, NY, USA, 2007. ACM.

[5] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 918–929. VLDB Endowment, 2006.

[6] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 131–140, New York, NY, USA, 2008. ACM.

[7] Mehran Sahami and Timothy D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, pages 377–386, New York, NY, USA, 2006. ACM.

[8] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 131–140, New York, NY, USA, 2007. ACM.

[9] Hanna Geppert, Martin Vogt, and Jrgen Bajorath. Current trends in ligand-based virtual screening: Molecular representations, data mining methods, new application areas, and performance evaluation. *Journal of Chemical Information and Modeling*, 50(2):205–216, 2010.

[10] George Karypis. Evaluation of item-based top-n recommendation algorithms. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, CIKM '01, pages 247–254, New York, NY, USA, 2001. ACM.

[11] Michael J Keiser, Bryan L Roth, Blaine N Armbruster, Paul Ernsberger, and Brian K Irwin, John J Shoichet. Relating protein pharmacology by ligand chemistry. *Nat Biotech*, 25(2):197–206, 2007.

[12] Florence L Stahura and Jurgen Bajorath. Virtual screening methods that complement hts. *Comb Chem High Throughput Screen*, 7(4):259–269, 2004.

[13] John Willett. *Similarity and Clustering in Chemical Information Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1987.

[14] Amit Chandel, Oktie Hassanzadeh, Nick Koudas, Mohammad Sadoghi, and Divesh Srivastava. Benchmarking declarative approximate selection predicates. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 353–364, New York, NY, USA, 2007. ACM.

[15] Dongjoo Lee, Jaehui Park, Junho Shim, and Sang-goo Lee. An efficient similarity join algorithm with cosine similarity predicate. In *Proceedings of the 21st International Conference on Database and Expert Systems Applications: Part II*, DEXA'10, pages 422–436, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[17] Andreas Hotho, Andreas Nürnberger, and Gerhard Paaß. A brief survey of text mining. *LDV Forum - GLDV Journal for Computational Linguistics and Language Technology*, 2005.

[18] David C. Anastasiu and George Karypis. L2ap: Fast cosine similarity search with prefix l-2 norm bounds. In *30th IEEE International Conference on Data Engineering*, ICDE '14, 2014.

[19] David C. Anastasiu and George Karypis. Efficient identification of tanimoto nearest neighbors. In *The 3rd IEEE International Conference on Data Science and Advanced Analytics*, DSAA '16, Under submission.

[20] Marzena Kryszkiewicz. Bounds on lengths of real valued vectors similar with regard to the tanimoto similarity. In Ali Selamat, NgocThanh Nguyen, and Habibollah Haron, editors, *Intelligent Information and Database Systems*, volume 7802 of *Lecture Notes in Computer Science*, pages 445–454. Springer Berlin Heidelberg, 2013.

[21] David C. Anastasiu and George Karypis. L2knng: Fast exact k-nearest neighbor graph construction with l2-norm pruning. In *24th ACM International Conference on Information and Knowledge Management*, CIKM '15, 2015.

[22] David C. Anastasiu and George Karypis. Pl2ap: Fast parallel cosine similarity search. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '15, pages 8:1–8:8, New York, NY, USA, 2015. ACM.

[23] David C. Anastasiu and George Karypis. Fast parallel cosine k-nearest neighbor graph construction. *To be submitted*, 2016.

[24] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[25] Joydeep Ghosh and Alexander Strehl. Similarity-based text clustering: A comparative study. In Jacob Kogan, Charles Nicholas, and Marc Teboulle, editors, *Grouping Multidimensional Data*, pages 73–97. Springer Berlin Heidelberg, 2006.

[26] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *J. Mach. Learn. Res.*, 5:361–397, December 2004.

[27] Venu Satuluri and Srinivasan Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *Proc. VLDB Endow.*, 5(5):430–441, January 2012.

[28] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[29] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *Proc. Internet Measurement Conf.*, 2007.

[30] Narender Singh, Rajarshi Guha, Marc A. Giulianotti, Clemencia Pinilla, Richard A. Houghten, and José L. Medina-Franco. Chemoinformatic analysis of combinatorial libraries, drugs, natural products, and molecular libraries small molecule repository. *Journal of Chemical Information and Modeling*, 49(4):1010–1024, 2009.

[31] Geroge Papadatos, Mark Davies, Nathan Dedman, Jon Chambers, Anna Gaulton, James Siddle, Richard Koks, Sean A. Irvine, Joe Pettersson, Nicko Goncharoff, Anne Hersey, and John P. Overington. Surechembl: a large-scale, chemically annotated patent document database. *Nucleic Acids Research*, 44:D1220–D1228, 2016.

[32] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[33] Nikil Wale, Ian A. Watson, and George Karypis. Indirect similarity based methods for effective scaffold-hopping in chemical compounds. *J. Chem. Info. Model*, 48:730–741, 2008.

[34] Nikil Wale and George Karypis. Acyclic subgraph based descriptor spaces for chemical compound retrieval and classification. In *Proceedings of the Sixth International Conference on Data Mining*, ICDM '06, 2006.

[35] Noel M. O'Boyle, Michael Banck, Craig A. James, Chris Morley, Tim Vandermeersch, and Geoffrey R. Hutchison. Open babel: An open chemical toolbox. *Journal of Cheminformatics*, 3(1):1–14, 2011.

[36] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 577–586, New York, NY, USA, 2011. ACM.

[37] Youngki Park, Sungchan Park, Sang-goo Lee, and Woosung Jung. Greedy filtering: A scalable algorithm for k-nearest neighbor graph construction. In *Database Systems for Advanced Applications*, volume 8421 of *Lecture Notes in Computer Science*, pages 327–341. Springer-Verlag, 2014.

[38] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 5–, Washington, DC, USA, 2006. IEEE Computer Society.

[39] Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proc. VLDB Endow.*, 1(1):933–944, August 2008.

[40] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. Top-k set similarity joins. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, ICDE '09, pages 916–927, Washington, DC, USA, 2009. IEEE Computer Society.

[41] Leonardo A. Ribeiro and Theo Härder. Efficient set similarity joins using min-prefixes. In *Proceedings of the 13th East European Conference on Advances in Databases and Information Systems*, ADBIS '09, pages 88–102, Berlin, Heidelberg, 2009. Springer-Verlag.

[42] Jiannan Wang, Guoliang Li, and Jianhua Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *Proceedings of the 2012*

ACM SIGMOD International Conference on Management of Data, SIGMOD '12, pages 85–96, New York, NY, USA, 2012. ACM.

[43] Amit Awekar and Nagiza F. Samatova. Fast matching for all pairs similarity search. In *Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology - Volume 01*, WI-IAT '09, pages 295–300, Washington, DC, USA, 2009. IEEE Computer Society.

[44] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[45] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers.

[46] Jiaqi Zhai, Yin Lou, and Johannes Gehrke. Atlas: a probabilistic algorithm for high dimensional similarity search. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 997–1008, New York, NY, USA, 2011. ACM.

[47] S. Joshua Swamidass, , and Pierre Baldi. Bounds and algorithms for fast exact searches of chemical fingerprints in linear and sublinear time. *Journal of Chemical Information and Modeling*, 47(2):302–317, 2007.

[48] Pierre Baldi, Daniel S. Hirschberg, and Ramzi J. Nasr. Speeding up chemical database searches using a proximity filter based on the logical exclusive or. *Journal of Chemical Information and Modeling*, 48(7):1367–1378, 2008.

[49] Ramzi Nasr, Daniel S. Hirschberg, and Pierre Baldi. Hashing algorithms and data structures for rapid searches of fingerprint vectors. *Journal of Chemical Information and Modeling*, 50(8):1358–1368, 2010.

[50] Thomas G. Kristensen, Jesper Nielsen, and Christian N. S. Pedersen. *Algorithms in Bioinformatics: 9th International Workshop, WABI 2009, Philadelphia, PA, USA,*

September 12-13, 2009. Proceedings, chapter A Tree Based Method for the Rapid Screening of Chemical Fingerprints, pages 194–205. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[51] Andrew Smellie. Compressed binary bit trees: A new data structure for accelerating database searching. *Journal of Chemical Information and Modeling*, 49(2):257–262, 2009.

[52] Yasuo Tabei and Koji Tsuda. Sketchsort: Fast all pairs similarity search for large databases of molecular fingerprints. *Molecular Informatics*, 30(9):801–807, 2011.

[53] Thomas G. Kristensen, Jesper Nielsen, and Christian N. S. Pedersen. Using inverted indices for accelerating lingo calculations. *Journal of Chemical Information and Modeling*, 51(3):597–600, 2011.

[54] Philipp Thiel, Lisa Sach-Peltason, Christian Ottmann, and Oliver Kohlbacher. Blocked inverted indices for exact clustering of large chemical spaces. *Journal of Chemical Information and Modeling*, 54(9):2395–2401, 2014.

[55] Thomas G. Kristensen. Transforming tanimoto queries on real valued vectors to range queries in euclidian space. *Journal of Mathematical Chemistry*, 48(2):287–289, 2010.

[56] Shereena M. Arif, John D. Holliday, and Peter Willett. Inverse frequency weighting of fragments for similarity-based virtual screening. *Journal of Chemical Information and Modeling*, 50(8):1340–1349, 2010.

[57] Leonardo Andrade Ribeiro and Theo Härder. Generalizing prefix filtering to improve set similarity joins. *Inf. Syst.*, 36(1):62–78, March 2011.

[58] Marzena Kryszkiewicz. Using non-zero dimensions for the cosine and tanimoto similarity search among real valued vectors. *Fundamenta Informaticae*, 127(1-4):307–323, 2013.

[59] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of*

the Twelfth International Conference on Information and Knowledge Management, CIKM '03, pages 426–434, New York, NY, USA, 2003. ACM.

[60] Trevor Strohman and W. Bruce Croft. Efficient document retrieval in main memory. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '07, pages 175–182, New York, NY, USA, 2007. ACM.

[61] Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 993–1002, New York, NY, USA, 2011. ACM.

[62] Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 113–122, New York, NY, USA, 2013. ACM.

[63] Cristian Rossi, Edleno S. de Moura, Andre L. Carvalho, and Altigran S. da Silva. Fast document-at-a-time query processing using two-tier indexes. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, pages 183–192, New York, NY, USA, 2013. ACM.

[64] Howard Turtle and James Flood. Query evaluation: Strategies and optimizations. *Inf. Process. Manage.*, 31(6):831–850, November 1995.

[65] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, January 2008.

[66] Jinyang Gao, Hosagrahar Visvesvaraya Jagadish, Wei Lu, and Beng Chin Ooi. Dsh: Data sensitive hashing for high-dimensional k-nn search. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1127–1138, New York, NY, USA, 2014. ACM.

[67] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 97–104, New York, NY, USA, 2006. ACM.

[68] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[69] Tamer Elsayed, Jimmy Lin, and Douglas W. Oard. Pairwise document similarity in large collections with mapreduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, HLT-Short '08, pages 265–268, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[70] Jimmy Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, pages 155–162, New York, NY, USA, 2009. ACM.

[71] G. De Francisci, C. Lucchese, and R. Baraglia. Scaling out all pairs similarity search with mapreduce. *Large-Scale Distributed Systems for Information Retrieval*, page 27, 2010.

[72] Ranieri Baraglia, Gianmarco De Francisci Morales, and Claudio Lucchese. Document similarity self-join with mapreduce. In *Proceedings of the 2010 IEEE International Conference on Data Mining*, ICDM '10, pages 731–736, Washington, DC, USA, 2010. IEEE Computer Society.

[73] Maha Ahmed Alabduljalil, Xun Tang, and Tao Yang. Optimizing parallel algorithms for all pairs similarity search. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pages 203–212, New York, NY, USA, 2013. ACM.

[74] Maha Alabduljalil, Xun Tang, and Tao Yang. Cache-conscious performance optimization for similarity search. In *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, pages 713–722, New York, NY, USA, 2013. ACM.

[75] Xun Tang, Maha Alabduljalil, Xin Jin, and Tao Yang. Load balancing for partition-based similarity search. In *Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, SIGIR '14, pages 193–202, New York, NY, USA, 2014. ACM.

[76] Amit Awekar and Nagiza F Samatova. Parallel all pairs similarity search. In *Proceedings of the 10th International Conference on Information and Knowledge Engineering*, IKE '11, 2011.

[77] Yu Jiang, Dong Deng, Jiannan Wang, Guoliang Li, and Jianhua Feng. Efficient parallel partition-based algorithms for similarity search and join with edit distance constraints. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 341–348, New York, NY, USA, 2013. ACM.

[78] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. Pass-join: A partition-based method for similarity joins. *Proc. VLDB Endow.*, 5(3):253–264, November 2011.

[79] Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, New York, NY, USA, 1999. ACM.

[80] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 233–244, New York, NY, USA, 2009. ACM.

[81] Aydin Buluç, Samuel Williams, Leonid Oliker, and James Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 721–733, Washington, DC, USA, 2011. IEEE Computer Society.

[82] Allstair Moffat, Ron Sacks-davis, Ross Wilkinson, and Justin Zobel. Retrieval of partial documents. In *Information Processing and Management*, pages 181–190, 1994.

[83] Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 743–754, New York, NY, USA, 2004. ACM.

[84] Chris Buckley and Alan F. Lewit. Optimization of inverted vector searches. In *Proceedings of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '85, pages 97–110, New York, NY, USA, 1985. ACM.

[85] Rodrigo Paredes, Edgar Chávez, Karina Figueroa, and Gonzalo Navarro. Practical construction of k-nearest neighbor graphs in metric spaces. In *Proceedings of the 5th International Conference on Experimental Algorithms*, WEA'06, pages 85–97, Berlin, Heidelberg, 2006. Springer-Verlag.

[86] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA, 2006. IEEE Computer Society.

[87] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961.

[88] Luiz DeRose, Bill Homer, and Dean Johnson. Detecting application load imbalance on high end massively parallel systems. In *Proceedings of the 13th International Euro-Par Conference on Parallel Processing*, Euro-Par'07, pages 150–159, Berlin, Heidelberg, 2007. Springer-Verlag.